Лабораторная работа №12. Основы веб-программирования на стороне сервера.

Добро пожаловать в мир веб-программирования! В течение этого курса я постараюсь вас научить, а вы, в свою очередь, постараетесь научиться создавать всевозможные Web-приложения, от элементарных примеров, до полнофункциональных продуктов.

Сразу замечу, что я пишу, рассчитывая, что вы знаете основы языка разметки HTML и имеете хотя бы небольшой опыт программирования. В противном случае... ну вы поняли. :)

Но, прежде, чем начать изучение непосредственно языка PHP, давайте разберемся, что же такое веб-программирование.

I. Что такое веб-программирование. Основные технологии

I-1. Клиент-Сервер

Если вы уже пробовали (а может быть, даже и небезуспешно :)) программировать, например, на Delphi, или Visual Basic, или даже Visual C++, то привыкли к такой схеме работы программы: нажимается кнопочка - выполняется код - выводится результат, и все это выполняется на одном компьютере.

В веб-программировании все обстоит по-другому.

Вы задумывались, что происходит, когда вы вводите в адресной строке браузера URL (Universal Resource Location, или в просторечии - адрес)? Схема работы следующая:

1. Браузер **открывает соединение** с сервером

2. Браузер отправляет серверу запрос на получение страницы

3. Сервер формирует ответ (чаще всего - HTML-код) браузеру и **закрывает соединение**

4. Браузер обрабатывает HTML-код и отображает страницу

Обратите внимание на выделенное **жирным**. Еще до того, как вы увидели на экране запрошенную страницу, соединение с сервером закрыто, и он о вас забыл. И когда вы введете другой (или тот же самый) адрес, или щелкните по ссылке, или нажмете на кнопку HTML-формы - та же схема повторится снова.

Такую схему работы называют **"клиент-сервер"**. Клиент в данном случае - браузер.

Итак, соединение с веб-сервером длится всего несколько секунд (или долей секунд) - это промежуток времени между щелчком по ссылке (или другим способом запроса) и началом отображения страницы. Большинство браузеров во время соединения отображают некий индикатор, например, MS Internet Explorer отображает анимацию в правом верхнем углу.

Внимательный читатель здесь может заметить - а как же так, я уже читаю страницу, а индикатор все еще показывает процесс соединения? Дело в том, что тэг <img src=...> (загрузка изображения) и некоторые другие - это не более чем еще один запрос к серверу - и выполняется он точно также, как и любой другой - по той же схеме. И запрос картинки, с точки зрения сервера, полностью независим от запроса HTML-ника.

Чтобы навсегда избавиться от восприятия HTTP как "черного ящика", "притворимся" браузером с помощью telnet'а:

1. Запустим **telnet www.php5.ru 80**

2. Введем в окне терминала следующее (если ввод не отображается - ничего страшного):

GET / HTTP/1.0 [здесь нажмем Enter]

Host: www.php5.ru [здесь нажмем Enter дважды]

Нажатие Enter соответствует, как правило, комбинации символов CR + LF, обозначаемых как **\r\n**. *Далее будет использоваться именно это*

обозначение.

По экрану пробежит HTML-код страницы http://www.php5.ru/. Как видите - ничего сложного.

Исходный код текущей страницы можно просмотреть практически в любом браузере, выбрав в меню "View|Source".

Картинки, фреймы - все это дополнительные запросы, точно такие же. Собственно, откуда берутся картинки в окне браузера: при парсинге (обработке) HTML-кода, браузер, натыкаясь на тэг <img src="*картинка*"> осуществляет дополнительный запрос к серверу - запрос *картинки*, и отображает ее на месте, где стоит тэг <img...>.

Попробуйте:

telnet www.php5.ru 80

GET /php/php5ru.png HTTP/1.0\r\n

Host: www.php5.ru\r\n\r\n

По экрану пробежит то, что вы увидите, если просмотрите этот png-файл в текстовом редакторе.

I-2. HTML-формы. Методы отправки данных на сервер

С HTML-формами вы наверняка уже встречались:

```
1.  <form method="GET" action="/cgi-bin/form_handler.cgi">
2.    Введите Ваше имя: <input type="text" name="name">
3.    <br>
4.    <input type="submit" name="okbutton" value="OK">
5.  </form>
```

Сохранив данный код в HTML-файле и просмотрев его с помощью вашего любимого браузера, вы увидите привычную HTML-форму:

Введите Ваше имя:

OK

Рассмотрим используемые в этом небольшом примере тэги подробнее.

Тэг **<form>**, имеющий парный завершающий тэг </form>, собственно и задает форму. Его атрибуты - оба необязательные:

• **action** - указывает URL (полный или относительный), на который будет **отправлена** форма. Отправка формы - это такой же запрос к серверу, как и все остальные (как я уже описал выше).

Если этот атрибут не указать, большинство браузеров (точнее говоря, все известные мне браузеры) отправляют форму на текущий документ, то есть "саму на себя". Это удобное сокращение, но по стандарту HTML атрибут action обязателен.

• **method** - ***способ*** отправки формы. Их два.

◦ **GET** - отправка данных формы в адресной строке. Вы могли заметить на различных сайтах присутствие в конце URL символа "?" и следующих за ним данных в формате *параметр=значение*. Здесь "параметр" соответствует *значению* атрибута **name** элементов формы (см. ниже про тэг **<input>**), а "значение" - содержимому атрибута **value** (в нем, например, содержится ввод пользователя в текстовое поле того же тэга **<input>**). Для примера - попробуйте поискать что-нибудь в **Яндексе** и обратите внимание на адресную строку браузера. Это и есть способ GET.

◦ **POST** - данные формы отправляются в **теле запроса**. Если не совсем понятно (или совсем непонятно), что это такое - не беспокойтесь, скоро мы к этому вопросу вернемся.

Если атрибут *method* не указан - подразумевается "GET".

Тэг **<input>** - задает **элемент формы**, определяемый атрибутом ***type***:

• Значение **"text"** задает однострочное текстовое поле ввода

• Значение **"submit"** задает кнопку, при нажатии которой происходит **отправка** формы на сервер

*Возможны и другие значения (да и **<input>** - не единственный тэг, задающий элемент формы), но их мы рассмотрим в следующих главах.*

Итак, что же происходит, когда мы нажимаем кнопку "OK"?

1. Браузер просматривает входящие в форму элементы и формирует из их атрибутов name и value **данные формы**. Допустим, введено имя **Vasya**. В этом случае данные формы - **name=Vasya&okbutton=OK**

2. Браузер устанавливает соединение с сервером, отправляет на сервер запрос документа, указанного в атрибуте **action** тэга <form>, используя метод отправки данных, указанный в атрибуте **method** (в данном случае - GET), передавая в запросе данные формы.

3. Сервер анализирует полученный запрос, формирует ответ, отправляет его браузеру и закрывает соединение

4. Браузер отображает полученный от сервера документ

Отправка того же запроса вручную (с помошью telnet) выглядит следующим образом (предположим, что доменное имя сайта - www.example.com):

telnet www.example.com 80

GET /cgi-bin/form_handler.cgi?name=Vasya&okbutton=OK HTTP/1.0\r\n

Host: www.example.com\r\n

\r\n

Как вы, скорее всего, уже догадались, нажатие submit-кнопки в форме с методом отправки "GET" аналогично вводу соответствующего URL (со знаком вопроса и данными формы в конце) в адресной строке браузера: **http://www.example.com/cgi-bin/form_handler.cgi?name=Vasya&okbutton=OK**

На самом деле, метод GET используется всегда, когда вы запрашиваете с сервера какой-либо документ, просто введя его URL, или щелкнув по ссылке. При использовании <form method="GET" ... >, к URL просто добавляются знак вопроса и данные формы.

Возможно, все эти технические подробности и упражнения с telnet-ом кажутся вам невероятно скучными и даже ненужными ("а при чем тут PHP?"). А зря. :) Это основы работы по протоколу HTTP, которые необходимо знать назубок каждому Web-программисту, и это не теоретические знания - все это пригодится на практике.

Теперь заменим первую строку нашей формы на следующую:

1. <form method="POST" action="/cgi-bin/form_handler.cgi">

Мы указали метод отправки "POST". В этом случае данные отправляются на сервер несколько другим способом:

telnet www.example.com 80

POST /cgi-bin/form_handler.cgi HTTP/1.0\r\n

Host: www.example.com\r\n

Content-Type: application/x-www-form-urlencoded\r\n

Content-Length: 22\r\n

\r\n

name=Vasya&okbutton=OK

При использовании метода POST данные формы отправляются уже после "двух Enter-ов" - в **теле** запроса. Все, что выше - на самом деле **заголовок** запроса (и когда мы использовали метод GET, данные формы отправлялись в заголовке). Для того, чтобы сервер знал, на каком байте закончить чтение тела запроса, в заголовке присутствует строка **Content-Length**; о том же, что данные формы будут переданы виде **параметр1=значение1&параметр2=значение2...**, причем значения передаются в виде urlencode - то есть, точно так же, как и с помощью метода GET, но в теле запроса, - серверу сообщает заголовок "Content-Type: application/x-www-form-urlencoded".

О том, что такое urlencode - чуть ниже.

Преимущество метода POST - отсутствие ограничения на длину строки с данными формы.

При использовании метода POST невозможно отправить форму, просто "зайдя по ссылке", как было с GET.

Для краткости изложения, введем термины "GET-форма" и "POST-форма", где префикс соответствует значению атрибута method тэга <form>.

*При использовании POST-формы, в ее атрибуте **action** можно указать после знака вопроса и параметры GET-формы. Таким образом, метод POST включает в себя и метод GET.*

I-3. Технология CGI

В предыдущей главе мы с вами разобрались, как создать HTML-форму, и как браузер отправляет введенные в нее данные на сервер. Но пока что непонятно, что будет сервер делать с этими данными.

Сам по себе веб-сервер умеет просто отдавать запрошенную страницу, и ничего более того, и ему все переданные данные формы, в общем-то, совершенно безразличны. Для того, чтобы можно было обработать эти данные с помощью какой-либо программы и **динамически** сформировать ответ броузеру, и была изобретена технология **CGI** (Common Gateway Interface).

Взглянем на этот URL: **http://www.example.com/cgi-bin/form_handler.cgi**. Первое предположение, которое можно сделать на его счет, обычно такое: сервер отдает содержимое файла form_handler.cgi из каталога cgi-bin. Однако, в случае с технологией CGI дело обстоит по-другому. Сервер **запускает программу** form_handler.cgi и передает ей данные формы. Программа же формирует текст, который передается браузеру в качестве ответа на запрос.

Программу form_handler.cgi можно написать на любом языке программирования, главное - соблюдать в программе стандарт CGI. Можно использовать, например, популярный скриптовый язык Perl. А можно написать все и на Си. Или на shell-скриптах... Но мы, для примера, напишем эту программу на Си. Но сначала разберемся, как происходит обмен данными между веб-сервером и CGI-программой.

- Перед запуском CGI-программы, сервер устанавливает **переменные окружения** (вам они наверняка знакомы по команде PATH). В каждый мало-мальски серьезном языке программирования есть средства для чтения переменных окружения. Стандарт CGI определяет весьма значительный набор переменных, которые должны быть определены перед запуском CGI-программы. Рассмотрим сейчас только три из них:

- **REQUEST_METHOD** - метод передачи данных - GET или POST (есть и другие, но пока мы их не рассматриваем)
- **QUERY_STRING** - содержит часть URL после вопросительного знака, или, другими словами, данные GET-формы.
- **CONTENT_LENGTH** - длина тела запроса (данные POST-формы).

• Сервер запускает CGI-программу. Тело запроса передается программе в виде **стандартного ввода** (stdin) - будто бы эти данные были введены с клавиатуры.

• Программа выдает ответ броузера на **стандартный вывод** (stdout) - "на экран". Этот вывод перехватывается веб-сервером и передается браузеру.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  // Читаем переменные среды, установленные веб-сервером
  char *query_string = getenv("QUERY_STRING");
  char *request_method = getenv("REQUEST_METHOD");

  char *post_data;      // Буфер для данных POST-запроса
  int post_length = 0;  // Длина тела запроса

  if (strcmp(request_method, "POST") == 0) {      // Если получен POST-запрос,
    post_length = atoi(getenv("CONTENT_LENGTH"));  // сначала читаем из
                                                   // переменной среды его длину,
    if (post_length) {                             // если она не нулевая,
      post_data = (char*)malloc(post_length+1);    // выделяем память для буфера,
      fread(post_data, post_length, 1, stdin);    // читаем со стандартного ввода тело запроса,
      post_data[post_length] = 0;                  // завершаем строку нулевым байтом.
```

```
20.       }
21.     }
22.
23.     // Выводим заголовок ответа...
24.     printf("Content-type: text/html\r\n\r\n");
25.
26.     // и его тело:
27.     printf("<h1>Здравствуйте!</h1>\r\n");
28.
29.     if (strlen(query_string)) {
30.       printf("<p>Параметры GET-формы: %s\r\n", query_string);
31.     }
32.
33.     if (post_length) {
34.       printf("<p>Параметры POST-формы: %s (длина тела запроса: %d)\r\n",
post_data, post_length);
35.       free(post_data); // не забываем освободить выделенную в строке 17 память
36.     }
37.
38.     return 0;
39.   }
```

Это простейшая CGI-программа на Си, выводящая содержимое полученных от веб-сервера параметров форм. Браузер в результате получит примерно следующий код (если "засабмитить" на эту программу POST-форму из последнего примера):

```
<h1>Здравствуйте!</h1>
<p>Параметры POST-формы: name=Vasya&okbutton=OK (длина тела запроса: 22)
```

Что при этом отобразится на экране пользователя, думаю, понятно без комментариев. :)

Как видите, даже простейшая программа вывода параметров не так-то проста. Более того, по стандарту HTTP почти все не алфавитно-цифровые символы (в т.ч. и русские буквы) передаются в так называемом UrlEncoded-виде (%XX, где XX - шестнадцатеричный код символа), и, если добавить в приведенную Си-программу код расшифровки UrlEncode, она уже не поместится на экран. А это - всего лишь базовые операции. А как вырастет программа на Си, если необходимо работать с базой данных?

Впрочем, написание CGI-программ на Си - довольно редкое извращение. :) Чаще всего это делают на Perl - языке, разработанном специально для обработки текстовых данных, а наличие модуля CGI делает написание CGI-скриптов намного более простой задачей. Здесь я не буду вас знакомить с Perl, отмечу лишь, что проблем остается достаточно: все же Perl не предназначен для Web, это язык универсальный. Да и сама технология CGI несовершенна: при каждом обращении происходит запуск программы (в случае с Perl - интерпретатор языка), а эта операция довольно ресурсоемкая: для домашней странички Васи Пупкина производительности, конечно, достаточно, но серьезный портал с десятками и сотнями тысяч хитов в сутки потребует уже огромных аппаратных мощностей.

А теперь взглянем на веб-сервер Apache. По своей природе он модульный, и позволяет подключать расширения добавлением одной строки в конфигурационный файл. (Ну, хорошо, хорошо, двух строк. :)). Было бы прекрасно, если бы существовал скриптовый язык, заточенный именно под Web, подключаемый модулем к Апачу, не так ли? Ну, вы уже поняли, к чему я клоню :) - это и есть **PHP**.

В принципе, PHP можно скомпилировать и как CGI-приложение, и использовать так же, как и Perl - но это для нестандартных веб-серверов или особых извращенцев. :)

II. PHP: Препроцессор Гипертекста

В 1994-м году, один программист, по имени Rasmus Lerdorf, намучавшись

с классическим перловым модулем CGI, решил написать несколько собственных Perl-скриптов, дабы было попроще создавать собственную домашнюю страницу, и назвал все это дело Personal Home Page (PHP). Через некоторое время ему понадобилось обрабатывать формы, ну и для увеличения производительности все было переписано на C - так появился Personal Home Page/Forms Interpreter (PHP/FI) 2.0. Труды свои Расмус, следуя принципам Open Source, выложил на всеобщее обозрение, и, в принципе, на некоторм количестве сайтов PHP/FI вполне успешно использовался, хотя был довольно примитивен.

В 1997-м на PHP/FI - в поисках инструмента для удобного Веб-скриптинга - наткнулись два других программера - Andi Gutmans и Zeev Suraski. Сама идея им понравилась, но функциональность и скорость работы PHP/FI оставляли желать лучшего, и Andi и Zeev решились переписать PHP с нуля. Язык получился универсальный и мощный, и вскоре привлек внимание множества веб-разработчиков: к концу 1998 года PHP3 использовался на ~10% веб-серверов. Скромное название "Personal Home Page" уже не очень-то соответствовало реальности, и название было изменено на - в лучших Unix-традициях - рекурсивное: **P**HP: **H**ypertext **P**reprocessor.

"Движок" PHP 4, названный Zend Engine, разрабатывался усилиями уже сформировавшегося и с тех пор непрерывно расрастающегося PHP community, и в 2000-м году вышла 4-я версия PHP, ставшая менее чем через полгода стандартом для Веб-разработки под Unix (и не только): каждый уважающий себя хостер предоставлял поддержку PHP. Сейчас подходит к концу разработка PHP5, основанного на новом Zend Engine 2...

Впрочем, хватит лирики. Давайте посмотрим на простой PHP-скрипт. Сначала немного изменим HTML-форму из предыдущего раздела:

```
<form method="POST" action="form_handler.php">
  Введите Ваше имя: <input type="text" name="name">
  <br>
  <input type="submit" name="okbutton" value="OK">
</form>
```

А теперь - form_handler.php:

```
<html>
<body>
<?
  echo "<h1>Привет, <b>" . $_POST['name'] . "</b></h1>!";
?>
</body>
</html>
```

В отличие от Си или Perl, php-скрипт представляет собой обычную, в общем-то, HTML-страницу: "просто так" написанные тэги передаются "как есть", будто бы это обычный html-ник. Сам скрипт заключается в специальные тэги **<?** и **?>**, внутри которых мы используем для вывода текста оператор echo. Таких блоков может быть сколько угодно, все, что между ними, интерпретируется как обычный html.

Переменные GET-запроса попадают в массив $_GET, переменные POST-запроса - в массив $_POST, серверные переменные (типа IP-адреса, имени скрипта и т.д.) - в $_SERVER, оператор "точка" (.) - объединение строк... Причем все служебные операции (чтение stdin и переменных среды, Url-декодирование) уже произвел сам PHP. Удобно, не так ли?

Далее. Зачем нам два файла - HTML с формой и PHP-скрипт? Вполне достаточно одного скрипта:

```
1.  <html>
2.  <body>
3.  <?
4.    if ($_SERVER['REQUEST_METHOD'] == 'POST') {
5.      echo "<h1>Привет, <b>" . $_POST['name'] . "</b></h1>!";
6.    }
7.  ?>
8.  <form method="POST">
9.    Введите Ваше имя: <input type="text" name="name">
10.   <br>
11.   <input type="submit" name="okbutton" value="OK">
12. </form>
13. </body>
14. </html>
```

Мы убрали из тэга form атрибут action - это значит, что форма отправляется "сама на себя", т.е. на текущий URL. Это иногда называют "postback form". В строке 4 с помощью оператора if проверяется, использовался ли для загрузки документа метод POST (аналог строки 13 примера на Си), и - если это так - в следующей строке выводится приветствие.

На этой простой программе - своего рода Web-варианте "Hello World".

Лабораторная работа №13. Основы синтаксиса PHP

Как выглядит PHP-программа

В отличие от традиционных скриптовых языков (таких, как Perl), PHP-программа представляет собой HTML-страницу со вставками кода. Для сравнения:

Perl-скрипт:

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";
print "<html>\n<head><title>Hello World</title></head>\n";
print "<body><h1>Hello World!</h1></body>\n";
print "</html>";
```

PHP-скрипт (да-да, это программа на PHP ;)):

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

Как видите, простейшая программа на PHP - это обычная HTML-страница. О выводе заголовка **Content-type: text/html** PHP тоже позаботился самостоятельно.

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

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

За распознавание тэга **<?** *как начала PHP-блока отвечает директива* ***short_open_tag*** *файла php.ini (по умолчанию - включена). Если вы хотите разрабатывать скрипты, работающие независимо от данной настройки, используйте длинный открывающий тэг* ***<?php****. Я буду использовать сокращенную форму.*

Рассмотрим простой пример.

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

Для выполнения примеров, скопируйте их в файл test.php, расположенный в каталоге, C:\WebServers\home\localhost\www, и выполните их, обратившись к сохраненному скрипту (test.php) из адресной строки браузера (http://localhost/test.php).

Если сегодня - 27-е июля 2004 года, в результате исполнения скрипта браузер получит следующий HTML-код:

```
<html>
<head><title>Hello World</title></head>
<body><h1>Hello World!</h1>
<p>Текущая дата:
27.07.2004</body>
</html>
```

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

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

Строка 6 является законченным выражением. Каждое выражение в PHP заканчивается точкой с запятой - **;**. Именно точкой с запятой, а не переводом строки - не забывайте об этом, особенно если вы раньше программировали на Visual Basic или ASP.

Внимательный читатель заметит, что тэг </body> расположен на той же строке, что и текст, сформированный функцией date(), хотя в исходном коде </body> находится на отдельной строке. Дело в том, что PHP отбрасывает перевод строки, следующий сразу после закрывающего тэга **?>** *- это сделано специально, чтобы в фрагментах HTML, где лишние пробелы нежелательны, не было необходимости жертвовать читабельностью скрипта, записывая закрывающий PHP-тэг на одной строке с последующим HTML-кодом. Если же пробел необходим - вставьте после* **?>** *пустую строку.*

Переменные и типы данных

Переменные в PHP начинаются со знака **$**, за которыми следует

произвольный набор латинских букв, цифр и знака подчеркивания: _, при этом цифра не может следовать сразу за знаком **$**.

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

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

Пример:

```
1.  <?
2.    $a = 'test';
3.    $copyOf_a  = $a;
4.    $Number100  = 100;
5.    echo $a;
6.    echo $copyOf_a;
7.    echo $Number100;
8.  ?>
```

Данный код выведет: **testtest100**.

*Следите за тем, какие имена вы даете переменным: вряд ли вы через полгода вспомните, для чего используется переменная **$a21** или **$zzz**. :) А вот для чего используется переменная **$username**, понять довольно легко.*

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

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

Как видите, в PHP существует типизация, то есть язык различает

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

Такое поведение (а также отсутствие необходимости (и даже возможности) явно определять переменные) роднит PHP с Perl и Basic, однако, возможно, будет встречено в штыки приверженцами строгих языков, таких как Си и Pascal. Конечно, необходимость четкого определения переменных и требование явного приведения типов уменьшает число возможных ошибок программирования, однако, PHP, прежде всего, - интерпретируемый язык для быстрой разработки скриптов, и нестрогость синтаксиса с лихвой компенсируется скоростью кодинга. А о неинициализированной переменной PHP всегда услужливо сообщит - если, конечно, ему этого не запрещать... Впрочем, я забегаю вперед.

Рассмотрим еще один пример:

```
1.  <?
2.      $greeting = 'Привет';
3.      $name = 'Вася';
4.      $message = "$greeting, $name!";
5.      echo $message;
6.  ?>
```

Особого внимания заслуживает четвертая строка. Внутри двойных кавычек указаны переменные, определенные в предыдущих строках. Если выполнить эту программу (вы ведь это уже сделали? :)), в окне браузера отобразится строка **Привет, Вася!**. Собственно, в этом и заключается основная особенность двойных кавычек: имена переменных, указанных внутри пары символов **"**, заменяются на соответствующие

этим переменным значения.

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

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

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

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

Экранирование (не обратный слэш, а сам принцип ;)) в PHP используется во многих случаях - так что с этим подходом мы еще не раз встретимся.

Если в данном кавычки заменить на одинарные, в браузере отобразится именно то, что внутри них написано (**$greeting, $name!**).

Комбинации символов, начинающиеся с \, в одинарных кавычках также никак не преобразуются, за двумя исключениями: \' - одинарная кавычка внутри строки; \\ - обратный слэш (в количестве одна штука :).

Немного изменим наш последний пример:

```
1.  <?
2.      $greeting = 'Привет';
3.      $name = 'Вася';
4.      $message = $greeting . ',' . $name. '!';
5.      echo $message;
6.  ?>
```

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

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

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

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

Кстати, в последних двух примерах совершенно нет необходимости в

определении переменной ***$message****: строки 4 и 5 можно сократить до* ***echo $greeting . ',' . $name . '!';****. Ну а если переменная* ***$message*** *нам может понадобиться в нижеследующем коде - можно написать* ***echo $message = $greeting . ',' . $name . '!';****, и это сработает. Связано это с тем, что результатом выражения, содержащего присваивние, является присвоенное значение. Это особенно удобно при присваивании одного и того же значения нескольким переменным. Например, если переменным* ***$a*** *и* ***$b*** *нужно присвоить одно и то же значение (скажем, число с плавающей запятой 10.34), можно написать* ***$a = $b = 10.34;****.*

В PHP предусмотрено множество встроенных функций для работы со строками. Описание их вы можете найти в официальной **документации**.

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

Условные операторы

if

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

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

Здесь используется оператор if..else - **условный оператор**. В общем виде он выглядит так:

if (условие)

выражение_1;

else

выражение_2;

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

`!=` - не равно;

`<` - меньше;

`>` - больше;

`<=` - меньше или равно;

`>=` - больше или равно.

Поэкспериментируйте, изменяя оператор сравнения и значения переменных. (Для логической правильности вывода на экран, потребуется, конечно, изменить и тексты, выводимые операторами echo :)).

Не путайте оператор сравнения `==` с оператором присваивания `=`! Если

вы допустите такую ошибку, условие всегда будет верным, если присваивается значение, соответствующее булевому true, и всегда ложным - если значение соответствует false. (См. выше о булевых значениях и чему они соответствуют).

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

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

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

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

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

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

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

Теперь все работает правильно.

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

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

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

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

Кстати, в строке 8 - комментарий. Это информация для человека, PHP ее игнорирует. Комментарии бывают двух видов: однострочный, как

здесь - начинается с `//` и распространяется до конца строки, и многострочный - комментарием считается все, что расположено между парами символов **`/*`** и **`*/`**.

Комментарий вида `//` - один из немногих случаев, когда инструкция заканчивается переводом строки. Напомню - PHP в большинстве случаев безразличны переводы строк: все предыдущие примеры вполне можно было бы записать в одну строку.

switch

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

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

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

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

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

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

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

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

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

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

Циклы

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

while

Начнем с цикла while:

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

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

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

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

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

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

Если бы мы не увеличвали значение **$i**, выход из цикла никогда бы

не произошел ("вечный цикл").

Запишем тот же пример в более краткой форме:

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

И еще один вариант:

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

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

do..while

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

Пример:

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

for

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

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

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

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

Запутались? ;) Перепишем "общий случай" цикла **for** в переложении на цикл **while**:

for	while

for (выражение_1; выражение_2; выражение_3) {
тело_цикла
}
	выражение_1;
	while (выражение_2) {
	тело_цикла
	выражение_3;
	}

Надеюсь, теперь все понятно. :) Точно понятно? Тогда разберитесь в этом цикле:

```php
<?
  $i=0;
  for ($i++; --$i<10; $i+=2) {
    echo $i . "<br>\n";
  }
?>
```

Если долго разбирались - ничего страшного:) Цикл for чаще всего используется в более понятной форме - как в первом примере.

Операторы break и continue. Вложенные циклы

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

```
1. <?
2.    $i = 0;
3.    while (++$i < 10) {
4.      echo $i . "<br>\n";
5.      if ($i == 5) break;
6.    }
7. ?>
```

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

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

```
1. <?
2.    for ($i=0; $i<10; $i++) {
3.      if ($i == 5) continue;
4.      echo $i . "<br>\n";
5.    }
6. ?>
```

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

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

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

Массивы

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

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

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

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

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

В других языках программирования (например, Perl) массивы, имеющие строковые индексы, называются ***хэшами (hash)****, и являются отдельным типом данных. В PHP же, по сути, все массивы являются хэшами, однако индексом может служить и строка, и число.*

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

```
1. <?
2.    $i = 1024;
3.    $a = array( 1=>'abc',  2=>100,  'test'=>$i-100  );
4.    print_r($a);
5. ?>
```

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

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

Строки в выводе функции print_r разделяются обычным переводом строки **\n***, но не тэгом
. Для удобства чтения, строку print_r(..) можно окружить операторами вывода тэгов <pre>...</pre>:*

```
echo '<pre>';
print_r($a);
echo '</pre>';
```

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

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

  echo "<pre>";
  print_r($operating_systems);
  echo "</pre>";
?>
```

Вывод:

Array

(

[0] => Windows

[1] => Linux

[2] => FreeBSD

[3] => OS/2

[4] => MS-DOS

)

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

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

  echo '<table border=1>';
  for ($i=0; $i<count($operating_systems); $i++) {
    echo '<tr><td>'.$i.'</td><td>'.$operating_systems[$i].'</td></tr>';
  }
  echo '</table>';
?>
```

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

индексов вручную.

```php
<?
  $languages = array(
    1 => 'Assembler',
    'C++',
    'Pascal',
    'scripting' => 'bash'
  );
  $languages['php'] = 'PHP';
  $languages[100] = 'Java';
  $languages[] = 'Perl';

  echo "<pre>";
  print_r($languages);
  echo "</pre>";
?>
```

Вывод:

Array

(

[1] => Assembler

[2] => C++

[3] => Pascal

[scripting] => bash

[php] => PHP

[100] => Java

[101] => Perl

)

Цикл foreach

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

```php
<?
  $languages = array(
    1 => 'Assembler',
    'C++',
    'Pascal',
    'scripting' => 'bash'
  );
  $languages['php'] = 'PHP';
  $languages[100] = 'Java';
  $languages[] = 'Perl';
?>
<table>
<tr>
  <th>Индекс</th>
  <th>Значение</th>
</tr>
<?
  foreach ($languages as $key => $value) {
    echo '<tr><td>' . $key . '</td><td>' . $value . '</td></tr>';
  }
?>
</table>
```

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

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

Конструкции list и each

В дополнение к уже рассмотренной конструкции **array**, существует дополняющая ее конструкция **list**, являющаяся своего рода антиподом **array**: если последняя используется для создания массива из набора

значений, то **list**, напротив, заполняет перечисленные переменные значениями из массива.

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

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

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

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

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

Эта особенность позволяет перебирать массив с помощью обычных

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

```
<?
  $browsers = array(
    'MSIE' => 'Microsoft Internet Explorer 6.0',
    'Gecko' => 'Mozilla Firefox 0.9',
    'Opera' => 'Opera 7.50'
  );

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

  ?>
```

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

Мы рассмотрели только самые основы работы с массивами. В PHP существует множество разнообразных функций работы с массивами; их подробное описание находится в соответствующем разделе **документации**.

Константы

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

```php
<?
  define ('MY_NAME', 'Вася');

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

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

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

Лабораторная работа №14. Формы

I. **HTML-формы. Массивы $_GET и $_POST**

II. **htmlspecialchars()**

III. **phpinfo()**

HTML-формы. Массивы $_GET и $_POST

Если вы еще не забыли материал **первой главы**, то вряд ли стоит напоминать: формы являются основным способом обмена данными между веб-сервером и браузером, то есть обеспечивают взаимодействие с пользователем - собственно, для чего и нужно веб-программирование.

*Для дальнейшего чтения обязательно **четкое** понимание описанных в **первой главе** основ веб-программирования. Если вы новичок, советую еще раз внимательно перечитать ее.*

Итак, возьмем уже знакомый вам по первой главе пример:

```
1.  <html>
2.  <body>
3.  <?
4.    if ($_SERVER['REQUEST_METHOD'] == 'POST') {
5.      echo '<h1>Привет, <b>' . $_POST['name'] . '</b></h1>!';
6.    }
7.  ?>
8.  <form method="POST" action="<?=$_SERVER['PHP_SELF']?>">
9.    Введите Ваше имя: <input type="text" name="name">
10.   <br>
11.   <input type="submit" name="okbutton" value="OK">
12. </form>
13. </body>
14. </html>
```

Форма, приведенная в строках 8-12, содержит два элемента: **name** и **okbutton**. Атрибут **method** указывает **метод отправки формы** POST, атрибут же action, указывающий URL, на который отправляется форма, заполняется значением серверной переменной PHP_SELF - адресом выполняемого в данный момент скрипта.

<?=$_SERVER['PHP_SELF']?> - сокращенная форма записи для echo: <? echo $_SERVER['PHP_SELF']; ?>.

Предположим, в поле name мы ввели значение **Вася**, и нажали кнопку **OK**. При этом браузер отправляет на сервер POST-запрос. Тело запроса: **name=Вася&okbutton=OK**. PHP автоматически заполняет массив **$_POST**:

$_POST['name'] = 'Вася'

$_POST['okbutton'] = 'OK'

*В действительности, значение "Вася" отправляется браузером в urlencode-виде; для кодировки windows-1251 это значение выглядит как **%C2%E0%F1%FF**. Но, поскольку PHP автоматически осуществляет необходимое декодирование, мы можем "забыть" об этой особенности - пока не придется работать с HTTP-запросами вручную.*

Так как в теле запроса указываются только имена и значения, но не типы элементов форм, PHP понятия не имеет, соответствует **$_POST['name']** строке ввода, кнопке, или списку. Но эта информация нам, в общем-то, совершенно не нужна. :)

Поскольку знать, что написано на кнопке submit, нам необязательно, в строке 11 можно удалить атрибут name, сократив описание кнопки до **<input type="submit" value="OK">**. В этом случае, браузер отправит POST-запрос **name=Вася**.

А теперь - то же самое, но для GET-формы:

```
1.  <html>
2.  <body>
3.  <?
4.    if (isset($_GET['name'])) {
5.      echo '<h1>Привет, <b>' . $_GET['name'] . '</b></h1>!';
6.    }
7.  ?>
8.  <form action="<?=$_SERVER['PHP_SELF']?>">
9.    Введите Ваше имя: <input type="text" name="name">
10.   <br>
11.   <input type="submit" value="OK">
12. </form>
13. </body>
14. </html>
```

В строке 8 можно было бы с таким же успехом написать **<form method="GET">**: GET - метод по умолчанию. В этот раз браузер отправляет GET-запрос, который равносилен вводу в адресной строке адреса: **http://*адрес-сайта*/*имя-скрипта*.php?name=Вася**.

PHP с GET-формами поступает точно так же, как и с POST, с тем отличием, что заполняется (угадайте с трех раз :)) массив **$_GET**.

Кардинальное же отличие - в строке 4. Поскольку простой ввод адреса в строке браузера является GET-запросом, проверка **if ($_SERVER['REQUEST_METHOD'] == 'GET')** бессмысленна: все, что мы в этом случае выясним - что кто-то от нефиг делать не отправил на наш скрипт POST-форму. ;) Поэтому мы прибегаем к конструкции **isset()**, которая возвращает **true**, если данная переменная определена (т.е. ей было присвоено значение), и **false** - если переменная не определена. Если форма была заполнена - как вы уже поняли, PHP автоматически присваивает $_GET['name'] соответствующее значение.

Способ проверки с помощью **isset()** - универсальный, его можно было бы использовать и для POST-формы. Более того, он предпочтительнее, так как позволяет выяснить, какие именно поля формы заполнены.

Во многих старых книгах и статьях утверждается, что:

1. Данные как из GET, так и из POST-форм попадают непосредственно в переменные (в нашем случае - $name), причем POST-данные приоритетнее, т.е. "затирают" GET-данные;
2. Также данные GET и POST-форм хранятся соответственно в массивах $HTTP_GET_VARS и $HTTP_POST_VARS.
Эта информация давным-давно устарела. Уже года три как. Web-программирование, и, в частности, PHP развивается быстрыми темпами. Запомните этот момент, чтобы не попасть впросак со старыми книгами или скриптами.

*Любознательные читатели могут узнать подробности **здесь**, а также проследовать **сюда**.*

Немного более сложный пример.

```
<html>
<body>
<?
  if (isset($_POST['name'], $_POST['year'])) {
    if ($_POST['name'] == '') {
      echo 'Укажите имя!<br>';
    } else if ($_POST['year'] < 1900 || $_POST['year'] > 2004) {
     echo 'Укажите год рождения! Допустимый диапазон значений: 1900..2004<br>';
    } else {
      echo 'Здравствуйте, ' . $_POST['name'] . '!<br>';
      $age = 2004 - $_POST['year'];
      echo 'Вам ' . $age . ' лет<br>';
    }
    echo '<hr>';
  }
?>
<form method="post" action="<?=$_SERVER['PHP_SELF']?>">
  Введите Ваше имя: <input type="text" name="name">
  <br>
  Введите Ваш год рождения: <input type="text" name="year">
  <input type="submit" value="OK">
</form>
</body>
</html>
```

Никаких новых приемов здесь не используется. Разберитесь, выполните код, попробуйте модифицировать...

Изменим последний пример, чтобы пользователю не нужно было повторно заполнять поля. Для этого заполним атрибуты value элементов формы только что введенными значениями.

```
1.  <html>
2.  <body>
3.  <?
4.    $name = isset($_POST['name']) ? $_POST['name'] : '';
5.    $year = isset($_POST['year']) ? $_POST['year'] : '';
6.
7.    if (isset($_POST['name'], $_POST['year'])) {
8.      if ($_POST['name'] == '') {
9.        echo 'Укажите имя!<br>';
10.     } else if ($_POST['year'] < 1900 || $_POST['year'] > 2004) {
11.       echo 'Укажите год рождения! Допустимый диапазон значений: 1900..2004<br>';
12.     } else {
13.       echo 'Здравствуйте, ' . $_POST['name'] . '!<br>';
14.       $age = 2004 - $_POST['year'];
15.       echo 'Вам ' . $age . ' лет<br>';
16.     }
17.     echo '<hr>';
18.   }
19. ?>
20. <form method="post" action="<?=$_SERVER['PHP_SELF']?>">
21.   Введите Ваше имя: <input type="text" name="name" value="<?=$name?>">
22.   <br>
23.   Введите Ваш год рождения: <input type="text" name="year" value="<?=$year?>">
24.   <input type="submit" value="OK">
25. </form>
26. </body>
27. </html>
```

Несколько непонятными могут оказаться строки 4 и 5. Все очень просто: ***X = A ? B : C*** - сокращенная запись условия **if (*A*) *X*=*B* else *X*=*C***. Строку 4 можно было бы записать так:

```
if (isset($_POST['name']))
  $name = $_POST['name'];
else
  $name = '';
```

Используемая же в строках 21 и 23 конструкция **<?=*$foo* ?>** - и того проще: это сокращение для **<? echo *$foo* ?>**.

Может возникнуть вопрос - почему бы не выбросить строки 4-5 и не написать:

**Введите Ваше имя: <input type="text" name="name" value="<?=$_POST['name']?>">
**

Введите Ваш год рождения: <input type="text" name="year" value="<?=$_POST['year']?>">

Дело в том, что, если эти POST-переменные не определены - а так и будет, если форму еще не заполняли, - PHP выдаст предупреждения об использовании неинициализированных переменных (причем, вполне обоснованно: такое сообщение позволяет быстро находить труднообнаружимые опечатки в именах переменных, а также предупреждает о возможных "дырах" на сайте). Можно, конечно, поместить код с isset... прямо в форму, но получится слишком громоздко. Разобрались? А теперь попробуйте найти ошибку в приведенном коде. Ну, не совсем ошибку, - но недочет.

htmlspecialchars()

Не нашли? Я подскажу. Введите, например, в поле "имя" двойную кавычку и какой-нибудь текст, например, **"123**. Отправьте форму, и взгляните на исходный код полученной страницы. В четвертой строке будет что-то наподобие:

Введите Ваше имя: <input type="text" name="name" value=""123">

То есть - ничего хорошего. А если бы хитрый пользователь ввел JavaScript-код?

Для решения этой проблемы необходимо воспользоваться функцией **htmlspecialchars()**, которая заменит служебные символы на их HTML-представление (например, кавычку - на **"**):

```
<html>
<body>
<?
  $name = isset($_POST['name']) ? htmlspecialchars($_POST['name']) : '';
  $year = isset($_POST['year']) ? htmlspecialchars($_POST['year']) : '';

  if (isset($_POST['name'], $_POST['year'])) {
    if ($_POST['name'] == '') {
      echo 'Укажите имя!<br>';
    } else if ($_POST['year'] < 1900 || $_POST['year'] > 2004) {
      echo 'Укажите год рождения! Допустимый диапазон значений: 1900..2004<br>';
    } else {
      echo 'Здравствуйте, ' . $name . '!<br>';
      $age = 2004 - $_POST['year'];
      echo 'Вам ' . $age . ' лет<br>';
    }
    echo '<hr>';
  }
?>
<form method="post" action="<?=$_SERVER['PHP_SELF']?>">
  Введите Ваше имя: <input type="text" name="name" value="<?=$name?>">
  <br>
  Введите Ваш год рождения: <input type="text" name="year" value="<?=$year?>">
  <input type="submit" value="OK">
</form>
</body>
</html>
```

Повторите опыт и убедитесь, что теперь HTML-код корректен.

Запомните - функцию **htmlspecialchars()** необходимо использовать всегда, когда выводится содержимое переменной, в которой могут присутствовать спецсимволы HTML.

phpinfo()

Функция **phpinfo()** - одна из важнейших в PHP. Она выводит информацию о настройках PHP, значения всевозможных конфигурационных переменных...

Почему я упоминаю о ней в главе, посвященной формам? **phpinfo()** - удобнейшее средство отладки. **phpinfo()**, помимо прочего, выводит

значения всех $_GET, $_POST и $_SERVER - переменных. Так что, если переменная формы "потерялась", самый простой способ обнаружить, в чем дело - воспользоваться функцией phpinfo. Для того, чтобы функция выводила **только** значения переменных (и вам не пришлось прокручивать десяток страниц), ее следует вызвать следующим образом: **phpinfo(INFO_VARIABLES);**, или - что абсолютно то же самое - **phpinfo(32);**.

Пример:

```
1.  <html>
2.  <body>
3.  <form method="post" action="<?=$_SERVER['PHP_SELF']?>">
4.    Введите Ваше имя: <input type="text" name="name">
5.    <input type="submit" value="OK">
6.  </form>
7.  <?
8.  phpinfo(32);
9.  ?>
10. </body>
11. </html>
```

Или, например, такая ситуация: вы хотите узнать IP-адрес посетителя. Вы помните, что соответствующая переменная хранится в массиве **$_SERVER**, но - вот незадача - забыли, как именно переменная называется. Опять же, вызываем **phpinfo(32);**, ищем в табличке свой IP-адрес и находим его - в строке **_SERVER["REMOTE_ADDR"]**.

Лабораторная работа №15. Функции

Определения и вызовы функций

Функция объявляется при помощи ключевого слова **function**, после которого в фигурных скобках записываются различные операторы, составляющие тело функции:

```
function MyFunction()
{
 // операторы
}
```

Если функция принимает аргументы, то они записываются как переменные в объявлении функции. **Аргумент функции** представляет собой переменную, передаваемую в тело функции для дальнейшего использования в операциях. В случае, когда функция принимает больше одного аргумента, эти переменные разделяются запятыми:

```
function MyFunction($var, $var1, $var2)
```

Если функция возвращает какое-либо значение, в теле функции обязательно должен присутствовать оператор **return:**

```
function MyFunction()
{
  return $ret; // возвращается значение переменной $ret
}
```

Пример простой функции.

```php
<?
  function get_sum()
  {
    $var = 5;
    $var1 = 10;
    $sum = $var + $var1;
    return $sum;
  }
  echo(get_sum()); // выводит 15
?>
```

В этом примере показана функция, вычисляющая сумму двух чисел. Эта функция не принимает ни одного аргумента, а просто вычисляет сумму и возвращает полученный результат. После этого, она вызывается в теле оператора **echo** для вывода результата в браузер. Модифицируем эту функцию так, чтобы она не возвращала полученный результат, а выводила его в браузер. Для этого достаточно внести оператор **echo** в тело функции:

```php
<?
  function get_sum()
  {
    $var = 5;
    $var1 = 10;
    $sum = $var + $var1;
    echo $sum;
  }
  get_sum();
?>
```

Переменные **$var** и **$var1** мы можем объявить как аргументы и в этом случае в теле функции их определять не надо:

```php
<?
  function get_sum($var, $var1)
  {
    $sum = $var + $var1;
    echo $sum;
  }
  get_sum(5,2); // выводит 7
?>
```

Переменная, содержащая значение, переданное через аргумент, называется **параметром** функции.

В рассмотренных примерах аргументы функции передаются по значению, т. е. значения параметров изменяется только внутри функции, и эти изменения не

влияют на значения переменных за пределами функции:

```
<?
  function get_sum($var) // аргумент передается по значению
  {
    $var = $var + 5;
    return $var;
  }
  $new_var = 20;
  echo(get_sum($new_var)); // выводит 25
  echo("<br>$new_var"); // выводит 20
?>
```

Для того чтобы переменные переданные функции сохраняли свое значение при выходе из нее, применяется передача параметров по ссылке. Для этого перед именем переменной необходимо поместить амперсанд (**&**):

```
function get_sum($var, $var1, &$var2)
```

В этом случае переменные **$var** и **$var1** будут переданы по значению, а переменная **$var2** - по ссылке. В случае, если аргумент передается по ссылке, при любом изменении значения параметра происходит изменение переменной-аргумента:

```
<?
  function get_sum(&$var) // аргумент передается по ссылке
  {
    $var = $var + 5;
    return$var;
  }
  $new_var = 20;
  echo(get_sum($new_var)); // выводит 25
  echo("<br>$new_var"); // выводит 25
?>
```

Область видимости переменных

Переменные в функциях имеют локальную область видимости. Это означает, что если даже локальная и внешняя переменные имеют одинаковые

имена, то изменение локальной переменной никак не повлияет на внешнюю переменную:

```
<?
  function get_sum()
  {
    $var = 5; // локальная переменная
    echo $var;
  }
  $var = 10; // глобальная переменная
  get_sum(); // выводит 5 (локальная переменная)
  echo("<br>$var"); // выводит 10 (глобальная
переменная)
?>
```

Локальную переменную можно сделать глобальной, если перед ее именем указать ключевое слово **global**. Если внешняя переменная объявлена как **global**, то к ней возможен доступ из любой функции:

```
<?
function get_sum()
  {
    global $var;
    $var = 5; // изменяем глобальную переменную
    echo $var;
  }
  $var = 10;
  echo("$var<br>"); // выводит 10
  get_sum(); // выводит 5 (глобальная переменная изменена)
?>
```

Доступ к глобальным переменным можно получить также через ассоциативный массив **$GLOBALS**:

```php
<?
  function get_sum()
  {
    $GLOBALS["var"] = 20; // изменяем глобальную переменную $var
    echo($GLOBALS["var"]);
  }
  $var = 10;
  echo("$var<br>"); // выводит 10
  get_sum(); // выводит 20 (глобальная переменная изменена)
?>
```

Массив **$GLOBALS** доступен в области видимости любой функции и содержит все глобальные перемене, которые используются в программе.

Время жизни переменной

Временем жизни переменной называется интервал выполнения программы, в течение которого она существует. Поскольку локальные переменные имеют своей областью видимости функцию, то время жизни локальной переменной определяется временем выполнения функции, в которой она объявлена. Это означает, что в разных функциях совершенно независимо друг от друга могут использоваться переменные с одинаковыми именами. Локальная переменная при каждом вызове функции инициализируется заново, поэтому функция-счетчик, в приведенном ниже примере всегда будет возвращать значение 1:

```php
function counter()
{
  $counter = 0;
  return ++$counter;
}
```

Для того, чтобы локальная переменная сохраняла свое предыдущее значение при новых вызовах функции, ее можно объявить статической при помощи ключевого слова **static**:

```php
function counter()
```

```
{
  static $counter = 0;
  return ++$counter;
}
```

Временем жизни статических переменных является время выполнения сценария. Т. е., если пользователь перезагружает страницу, что приводит к новому выполнению сценария, переменная **$counter** в этом случае инициализируется заново.

Что такое рекурсия

Рекурсией называется такая конструкция, при которой функция вызывает саму себя. Различают прямую и косвенную рекурсии. Функция называется прямо рекурсивной, если содержит в своем теле вызов самой себя. Если же функция вызывает другую функцию, которая в свою очередь вызывает первую, то такая функция называется косвенно рекурсивной.

Рассмотрим классические примеры использования рекурсии - реализацию операции возведения в степень и вычисление факториала числа. Заметим, что эти примеры являются классическими только из-за их удобства для объяснения понятия рекурсии, однако они не дают выигрыша в программной реализации по сравнению с итерационным способом решения этих задач.

```
<?
  function degree($x,$y)
  {
    if($y)
    {
      return $x*degree($x,$y-1);
    }
    return 1;
  }
  echo(degree(2,4)); // выводит 16
?>
```

Этот пример основан на том, что x^y эквивалентно $x*x^{(y-1)}$. В этом коде задача вычисления 2^4 разбивается на вычисление $2*2^3$. Затем $2*2^3$ разбивается на $2*2^2$ и так до тех пор, пока показатель не станет равным нулю.

Итерационный вариант этого примера выглядит так:

```
<?
  function degree($x,$y)
  {
    for($result = 1; $y > 0; --$y)
    {
      $result *= $x;
    }
    return $result;
  }
  echo(degree(2,4)); // выводит 16
?>
```

Кроме того, что этот код намного легче понять, он еще и более эффективен, поскольку проход цикла обходится "дешевле" вызова функции.

```
<?
  function fact($x)
  {
    if ($x < 0) return 0;
    if ($x == 0) return 1;
    return $x * fact($x - 1);
  }
  echo (fact(3)); // выводит 6
?>
```

Для отрицательного аргумента функция возвращает нулевое значение, так как факториал отрицательного числа не существует по определению. Для нулевого параметра функция возвращает значение 1, поскольку 0! = 1. В иных случаях вызывается та же функция с уменьшенным на 1 значением параметра, после чего результат умножается на текущее значение параметра. Т. е. происходит вычисление произведения:

```
k * (k - 1) * (k - 2) * ... * 3 * 2 * 1 * 1
```

Последовательность рекурсивных вызовов прерывается только при вызове **fact(0)**, который и приводит к последнему значению 1 в произведении,

так как последнее выражение, из которого вызывается функция, имеет вид 1 * **fact**(1 - 1).

Итерационно факториал можно вычислить так:

```
<?
  function fact($x)
  {
    for ($result = 1; $x > 1; --$x)
    {
      $result *= $x;
    }
    return $result;
  }
  echo (fact(6)); // выводит 720
?>
```

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

Вариант № 1

1. Дано натуральное число n > 100. Вывести на экран фразу «Мне n лет», учитывая, что при некоторых значениях n слово «лет» надо заменить на слово «год» или «года».

2. Дано число m (1 ≤ m ≤ 12).Определить, сколько дней в месяце с номером m.

Вариант № 2

1. Дано число m (1 ≤ m ≤ 7). Вывести на экран название дня недели, который соответствует этому номеру.

2. С клавиатуры вводится натуральное число n. В зависимости от значения остатка r при делении числа n на 7 вывести на экран число n в виде n = 7*k + r.

Вариант № 3

1. С клавиатуры вводится цифра m (от 1 до 4). Вывести на экран названия месяцев, соответствующих времени года с номером m (считать зиму временем года № 1).

2. Дано целое число C такое, что |C| < 9. Вывести это число в словесной форме, учитывая его знак.

Вариант № 4

1. С клавиатуры вводится цифра m (от 1 до 12). Вывести на экран название месяца, соответствующего цифре,

2. Дано число m (1≤ m ≤ 12). Определить полугодие, на которое приходится месяц с номером m и количество дней в том месяце (год не високосный).

Вариант № 5

1. Вводится число экзаменов N<=20. Напечатать фразу "Мы успешно сдали N экзаменов", согласовав слово "экзамен" с числом N.

2. Вводится число карандашей N<=10. Вывести фразу "Я купил N карандашей, согласовав слово "карандаш" с числом N.

Вариант № 6

1. Компания по снабжению электроэнергией взимает плату с клиентов по тарифу:

7 р. за 1 кВт/ч за первые 250 кВт/ч;

17 р. за кВт/ч, если потребление свыше 250, но не превышает 300 кВт/ч;

20 р. за кВт/ч, если потребление свыше 300 кВт/ч.

Потребитель израсходовал n кВт/ч. Подсчитать плату.

2. При покупке товара на сумму от 200 до 500 руб предоставляется скидка 3%, при покупке товара на сумму от 500 до 800 - скидка 5%, при покупке товара на сумму от 800 до 1000 руб - скидка 7%, свыше 1000 руб - скидка 9%. Покупатель приобрел 8 рулонов обоев по цене X1 и две банки краски по цене X2. Сколько он заплатил?

Лабораторная работа № 16. Работа с файлами

Открытие файлов

Файл представляет собой последовательность байтов, хранящуюся на каком-либо физическом носителе информации. Каждый файл имеет абсолютный путь, по которому определяется его местонахождение. В качестве разделителя пути в Windows может использоваться как прямой (/), так и обратный (\) слеш. В других операционных системах используется только прямой слеш.

Открытие файлов в файловой системе сервера производится при помощи функции **fopen**:

```
int fopen(string filename, string mode [, int use_include_path])
```

Первый аргумент **filename** - имя файла или абсолютный путь к нему. Если абсолютный путь не указывается, то файл должен находится в текущем каталоге.

Второй аргумент **mode** говорит о том, для каких действий открывается файл и может принимать следующие значения:

- **r** (Открыть файл только для чтения; после открытия указатель файла устанавливается в начало файла);
- **r+** (Открыть файл для чтения и записи; после открытия указатель файла устанавливается в начало файла);
- **w** (Создать новый пустой файл только для записи; если файл с таким именем уже есть вся информация в нем уничтожается);
- **w+** (Создать новый пустой файл для чтения записи; если файл с таким именем уже есть вся информация в нем уничтожается);
- **a** (Открыть файл для дозаписи; данные будут записываться в конец файла);
- **a+** (Открыть файл для дозаписи и чтения данных; данные будут записываться в конец файла);

- **b** (Флаг, указывающий на работу (чтение и запись) с двоичным файлом; указывается только в Windows).

Третий необязательный аргумент **use_include_path** определяет должны ли искаться файлы в каталоге **include_path**. (Параметр **include_path** устанавливается в файле php.ini).

В случае удачного открытия файла, функция **fopen** возвращает дескриптор файла, в случае неудачи - **false**. **Дескриптор файла** представляет собой указатель на открытый файл, который используется операционной системой для поддержки операций с этим файлом. Возвращенный функцией дескриптор файла необходимо затем указывать во всех функциях, которые в дальнейшем будут работать с этим файлом.

Код, приведенный ниже, открывает файл C:/WWW/HTML/file.txt для чтения:

```
<?
  $file = fopen("c:/www/html/file.txt","r");
  if(!file)
    {
      echo("Ошибка открытия файла");
    }
```

Открытие двоичного файла, к примеру, рисунка происходит таким же образом, только с флагом **b**:

```
<?
  $file = fopen("c:/www/html/river.jpg","rb");
  if(!file)
    {
      echo("Ошибка открытия файла");
    }
?>
```

Отображение файлов

Содержимое открытого файла можно отобразить в браузере с помощью функции **fpassthru**:

```
int fpassthru (int file)
```

Аргумент **file** представляет собой дескриптор файла.

```
<?
  $file = fopen("c:/www/html/pavlovo.jpg","rb");
  if(!file)
     {
       echo("Ошибка открытия файла");
     }
     else
     {
       fpassthru($file);
     }
?>
```

Для текстовых файлов существует еще одна функция отображения **readfile**:

```
  readfile (string filename)
```

Обращаем ваше внимание на то, что в качестве аргумента эта функция принимает имя файла, а не его дескриптор:

```
<?
  readfile ("file.txt");
?>
```

Закрытие файлов

После того, как вы закончите работу с файлом его необходимо закрыть. Закрытие файлов осуществляется с помощью функции **fclose**:

```
  int fclose (int file)
```

Аргумент **file** представляет собой дескриптор файла, который необходимо закрыть.

Чтение из файлов и запись в файлы

Чтение из файлов

Прочитать строку из открытого файла можно с помощью функции **fread**:

```
string fread ( int file, int length )
```

Эта функция возвращает строку длиной **length** символов из файла с дескриптором **file**.

Пример (чтение из файла):

```
<?
  $file = fopen("c:/www/html/file.txt","r");
  if(!file)
    {
      echo("Ошибка открытия файла");
    }
    else
    {
      $buff = fread ($file,100);
      print $buff;
    }
?>
```

Для чтения из файла можно также пользоваться функцией **fgets**:

```
string fgets ( int file, int length)
```

Эта функция читает и возвращает строку длиной **length - 1** байт. Чтение прекращается, когда достигнута новая строка или конец файла. При достижении конца файла функция возвращает пустую строку.

Для чтения файла с удалением из него тегов HTML применяется функция **fgetss**:

```
string fgetss (int file, int length [, string allowable_tags])
```

Необязательный третий параметр **allowable_tags** может содержать строку со списком тегов, которые не должны быть отброшены, при этом теги в строке записываются через запятую.

Если необходимо записать содержимое файла в массив, применяется функция **file**:

```
array file (string filename [, int use_include_path])
```

Функция считывает файл с именем **filename** и возвращает массив, каждый элемент которого соответствует строке в прочитанном файле. В следующем примере с помощью функции читается файл, информация из которого затем выводится в браузер.

```
<?
  $file_array = file("file.txt");
  if(!$file_array)
  {
    echo("Ошибка открытия файла");
  }
  else
  {
    for($i=0; $i < count($file_array); $i++)
    {
      printf("%s<br>", $file_array[$i]);
    }
  }
?>
```

Эта функция удобна также тем, что с ее помощью можно легко подсчитать количество строк в файле:

```php
<?
  $file_array =  file ("file.txt");
  if(!$file_array)
  {
    echo("Ошибка открытия файла");
  }
  else
  {
    $num_str =  count($file_array);
    echo($num_str);
  }
?>
```

Заметим, что функцию **file** следует применять лишь для чтения небольших файлов.

Для чтения файлов с расширением **.csv* применяется функция **fgetcsv**:

```php
array fgetcsv ( int file, int length, char delim)
```

Функция читает строку из файла и разбивает ее по символу **delim**. Параметр delim должен обязательно быть строкой из одного символа, иначе принимается во внимание только первый символ этой строки. Функция возвращает получившийся массив или false, если достигнут конец файла. Пустые строки в файле не игнорируются, а возвращаются как массив из одного элемента - пустой строки. Параметр **length** задает максимальную длину строки точно так же, как это делается в функции **fgets**.

Формат CSV является одним из форматов, в котором может сохранять файлы MSExcel. В следующем примере производится чтение созданного MSExcel файла file.csv, содержащего пароли пользователей.

```php
<?
  $count = 1;
  $file =  fopen ("file.csv","r");
  while ($data = fgetcsv ($file, 1000, ","))
  {
    $num = count ($data);
    $count++;
    for ($i=0; $i < $num; $i++)
    {
      print "$data[$i]<br>";
    }
  }
  fclose ( $file );
?>
```

Запись в файлы

Запись в файлы осуществляется функциями **fputs** и **fwrite**, которые абсолютно идентичны:

```
int fputs ( int file, string string [, int length ])
int fwrite ( int file, string string [, int length ])
```

Первый аргумент - дескриптор файла, в который осуществляется запись. Второй аргумент представляет собой строку, которая должна быть записана в файл. Третий необязательный аргумент задает количество символов в строке, которые должны быть записаны. Если третий аргумент не указан, записывается вся строка.

В этом примере в файл "file.txt" записывается строка "Hello, world!"

```php
<?
  $file = fopen ("file.txt","r+");
  $str = "Hello, world!";
  if ( !$file )
  {
    echo("Ошибка открытия файла");
  }
  else
  {
    fputs ( $file, $str);
  }
  fclose ($file);
?>
```

Копирование, переименование и удаление файлов

Копирование файлов осуществляется функцией **copy**:

```
int copy ( string file1, string file2)
```

Функция копирует файл с именем **file1** в файл с именем **file2**. Если файл **file2** на момент копирования существовал, то он перезаписывается.

Переименование файла производится с помощью функции **rename**:

```
int rename ( string old, string new)
```

Эта функция переименовывает файл с именем **old** в файл с именем **new**.

Функция **rename** не выполняет переименования файла, если его новое имя расположено в другой файловой системе.

Удаление файла осуществляется посредством функции **unlink**:

```
int unlink ( string filename)
```

Атрибуты файлов

Для получения дополнительной информации об атрибутах файла вы можете

воспользоваться перечисленными ниже функциями.

Функция **file_exists** проверяет, существует ли файл и возвращает true, если файл существует и false в противном случае:

```
bool file_exists ( string filename)
```

Функция **fileatime** возвращает время последнего обращения к файлу:

```
int fileatime ( string filename)
```

Функция **filemtime** возвращает время последней модификации содержимого файла:

```
int filemtime ( string filename)
```

Функция **file_size** возвращает размер файла в байтах:

```
int file_size ( string filename)
```

Функция **file_type** возвращает тип файла:

```
string file_type ( string filename)
```

Строка, возвращаемая этой функцией, содержит один из следующих типов файла:

- char (специальное символьное устройство);
- dir (каталог);
- fifo (именованный канал);
- link (символическая ссылка);
- block (специальное блочное устройство);
- file (обычный файл);
- unknown (тип не установлен).

Поскольку использование функций, возвращающих характеристики файла, весьма ресурсоемко, во избежание потери производительности при вызовах таких функций, PHP кэширует информацию о файле. Очистить этот кэш можно с помощью функции **clearstatcache**:

```
<?
clearstatcache();
```

```
?>
```

Перемещение по файлам

При чтении данных из файла указатель текущей позиции перемещается к очередному непрочитанному символу. Существует несколько функций, с помощью которых можно управлять положением этого указателя.

Установка указателя текущей позиции в начало файла производится функцией **rewind**:

```
int rewind ( int file)
```

Аргумент **file** является дескриптором файла.

Узнать текущее положение указателя можно при помощи функции **ftell**:

```
int ftell ( int file)
```

Установить указатель в любое место файла можно, используя функцию **fseek**:

```
int fseek ( int file, int offset [, int whence ])
```

Функция **fseek** устанавливает указатель файла на байт со смещением **offset** (от начала файла, от его конца или от текущей позиции, в **зависимости от значения параметра whence**). Аргумент **file** представляет собой дескриптор файла. Аргумент **whence** задает с какого места отсчитывается смещение **offset** и может принимать одно из следующих значений:

- **SEEK_SET** (отсчитывает позицию начала файла);
- **SEEK_CUR** (отсчитывает позицию относительно текущего положения указателя);
- **SEEK_END** (отсчитывает позицию относительно конца файла).

По умолчанию аргумент **whence** имеет значение SEEK_SET.

Узнать, находится ли указатель в конце файла, можно с помощью функции **feof**:

```
int feof ( int file)
```

Если указатель находится в конце файла, функция возвращает **true**, в ином случае возвращается false.

Функцию **feof** удобно использовать при чтении файла:

```
<?
  $file = fopen ("file.txt","r");
  if ($file)
  {
    while(!feof($file))
    {
      $str = fgets($file);
      echo $str;
      echo ("<br>");
    }
    fclose ( $file);
  }
  else
  {
    echo("Ошибка открытия файла");
  }
?>
```

При помощи этой функции удобно также определять количество строк в файле:

```
<?
  $file = fopen ("file.txt","r");
  if ($file)
  {
    $counter = 0;
    while(!feof($file))
    {
      $str = fgets ($file);
      $counter++;
    }
    echo($counter);
    fclose ($file);
```

```
  }
  else
  {
    echo("Ошибка открытия файла");
  }
?>
```

Работа с каталогами

Для установки текущего каталога применяется функция **chdir**:

```
  int chdir ( string directory)
```

Работать с этой функцией можно следующим образом:

- **chdir(**"/tmp/data"**);** // переход по абсолютному пути
- **chdir(**"./js"**);** // переход в подкаталог текущего каталога
- **chdir(**".."**);** // переход в родительский каталог
- **chdir(**"~/data"**);** // переходим в /home/пользователь/data (для Unix)

Чтобы узнать текущий каталог можно воспользоваться функцией **getcwd**:

```
  string getcwd ( string path)
```

Для того чтобы открыть каталог используется функция **opendir**, открывающая каталог, заданный параметром path:

```
  int opendir ( string path)
```

После того, как каталог открыт, прочитать его можно функцией **readdir**:

```
  string readdir ( int dir)
```

Эта функция возвращает имена элементов, содержащихся в каталоге. Кроме файлов и папок в каталогах находятся также элементы "." и "..". Первый элемент указывает на текущий каталог, а второй - на родительский. Текущий каталог, кстати, можно открыть, указав его имя как ".":

```
   $dir = opendir (".");
```

После того, как работа с каталогом закончена, его нужно закрыть. Закрытие каталога выполняется при помощи функции **closedir**:

```
  void closedir ($dir)
```

Ниже приведен пример, осуществляющий чтение и вывод файлов, находящихся в текущем каталоге.

```
<?
  $dir = opendir (".");
  echo "Files:\n";
  while ($file = readdir ($dir))
  {
    echo "$file<br>";
  }
  closedir ($dir);
?>
```

Заметим, что эта функция возвращает также "." и "..". Если этого делать не нужно, то исключить эти значения можно следующим образом:

```
<?
  $dir = opendir (".");
  while ( $file = readdir ($dir))
  {
    if (( $file != ".") && ($file != ".."))
    {
      echo "$file<br>";
    }
  }
  closedir ($dir);
?>
```

В качестве примера на рассмотренные функции, давайте создадим скрипт, удаляющий все файлы из каталога c:/temp, к которым не было доступа в течение суток. Функция удаления файлов в этом случае вызывается рекурсивно.

```
<?
  function delTemporaryFiles ($directory)
  {
  $dir = opendir ($directory);
  while (( $file = readdir ($dir)))
  {
    if( is_file ($directory."/".$file))
    {
      $acc_time = fileatime ($directory."/".$file);
      $time =  time();
      if ((($time - $acc_time) > 24*60*60)
      {
        if (  unlink ($directory."/".$file))
        {
          echo ("Файлы успешно удалены");
        }
      }
    }
    else if ( is_dir ($directory."/".$file) && ($file != ".") &&
($file != ".."))
    {
      delTemporaryFiles ($directory."/".$file);
    }
  }
  closedir ($dir);
  }
  delTemporaryFiles ("c:/temp");
?>
```

Создание каталогов производится с помощью функции **mkdir**:

```
  bool mkdir ( string dirname, int mode)
```

Эта функция создает каталог с именем dirname и правами доступа mode. В случае неудачи возвращает false. Права доступа задаются только для каталогов UNIX, поскольку в Windows этот аргумент игнорируется. Ниже

приведен пример создания каталога test в директории c:/temp.

```
<?
  $flag = mkdir ("c:/temp/test", 0700);
  if($flag)
  {
    echo("Каталог успешно создан");
  }
  else
  {
    echo("Ошибка создания каталога");
  }
?>
```

Удалить каталог можно с помощью функции **rmdir**:

```
  bool rmdir ( string dirname)
```

Теперь удалим только что созданный каталог /test:

```
<?
  $flag = rmdir ("c:/temp/test");
  if($flag)
  {
    echo("Каталог успешно удален");
  }
  else
  {
    echo("Ошибка удаления каталога");
  }
?>
```

Функция **rmdir** удаляет только пустые каталоги. Для того чтобы удалять непустые каталоги, давайте напишем функцию и удалим каталог c:/temp со всеми вложенными папками и файлами:

```
<?
  function full_del_dir ($directory)
```

```
  {
  $dir = opendir($directory);
  while(($file = readdir($dir)))
  {
    if ( is_file ($directory."/".$file))
    {
      unlink ($directory."/".$file);
    }
    else if ( is_dir ($directory."/".$file) &&
              ($file != ".") && ($file != ".."))
    {
      full_del_dir ($directory."/".$file);
    }
  }
  closedir ($dir);
  rmdir ($directory);
  echo("Каталог успешно удален");
  }
  full_del_dir ("c:/temp")
?>
```

При рекурсивном вызове функции не передавайте в качестве аргументов записи "." и "..", указывающие на текущий и родительский каталоги, так как в этом случае вы можете потерять ваши данные. Пропускайте эти записи явным образом при помощи условного оператора.

Методы PUT и POST

Методы HTTP PUT и HTTP POST предназначены для загрузки файлов на сервер.

Протокол HTTP предоставляет три метода для работы с информацией, находящейся на Web-сервере: GET, PUT и POST. Метод GET применяется для получения Web-страниц, при этом все переменные формы передаются в URL. Поскольку на многих Web-серверах установлено ограничение на

максимальную длину URL (как правило, не более 1024), не стоит применять метод GET, если требуется передача данных большего объема.

Метод PUT применяется для обновления информации на сервере, и требует, чтобы содержимое запроса HTTP PUT сохранялось на сервере. Запрос выглядит таким образом:

```
PUT /path/filename.html HTTP/1.1
```

В этом случае Web-сервер должен сохранить содержимое этого запроса в виде /path/filename.html в пространстве имен URL Web-сервера. По умолчанию сам Web-сервер не выполняет такие запросы, а задает CGI-сценарий для их обработки. В Apache назначить сценарий для обработки PUT-запросов, можно изменив директиву Script, находящуюся в файле httpd.conf, к примеру, так:

```
Script PUT /cgi-bin/put.cgi
```

Это означает, что обрабатывать PUT-запросы будет CGI-скрипт put.cgi.

Как правило, для загрузки файлов на сервер используют метод HTTP POST. Этот метод позволяет передавать большие объемы данных из формы и сохраняет все переменные формы в теле запроса.