**Раздел 4.**

**WEB-программирование на стороне сервера**

**4.1 Установка Apache и PHP**

**4.2 Основы синтаксиса PHP. Функции и области видимости**

**4.3 Работа с массивами и строками**

**4.4 Стандартные функции PHP для работы с файловой структурой и управления временем. Управление сессиями**

**4.5 Работа с базами данных MySQL**

**4.1 Установка Apache и PHP**

Самый распространенный Web - сервер в мире – это Apache. Последнюю версию Apache можно загрузить с сервера http://httpd.apache.org.

**Установка Web-сервера**

Для установки http-сервера Apache в операционной системе Windows загрузите дистрибутив ***apache\_x.x.xx-win32-x86-no\_ssl.msi*** с веб-узла <http://httpd.apache.org>. В данном случае ***x.x.xx*** – обозначает версию дистрибутива. Расширение «***msi****»* указывает, что для процедуры инсталляции потребуется программа инсталляции «Microsoft Installer», по умолчанию входящая в состав операционной системы семейства Windows.

Двойным щелчком по файлу дистрибутива запустите процесс инсталляции http-сервера Apache. В случае если MS Installer отсутствует или не поддерживает формат дистрибутива, может потребоваться обновить программу MS Installer. Это можно сделать путем загрузки соответствующего обновления с web-сервера компании Microsoft ([www.microso](http://www.microso) ft.com).

Если все в порядке, на экране появится приветственное окно мастера установки web-сервера. Нажмите кнопку «**NEXT**». В следующем окне ознакомьтесь с лицензионным соглашением и примите его. Следующее окно выводит информацию о продукте Apache HTTP Server. Нажмите «**NEXT**»и перейдите к окну настройки параметров Вашего будущего сервера (рис.1).



Рис. 1.Пример заполнения информации о сервере

В качестве имени сервера введите «**localhost**». Имя «localhost» является синонимом IP-адреса 127.0.0.1, который предназначен для замыкания передаваемых пакетов на себя.

Следующее окно позволяет выбрать режим установки «**Typical**» (обычный) или «**Custom**» (пользовательский). Для того чтобы иметь возможность установить сервер в папку отличную от папки по умолчанию, выберите вариант пользовательской установки.



Рис***. 2.*** Выбор пользовательских предпочтений

В качестве целевой папки установки Web - сервера лучше использовать не папку «**C:\Program Files**», создаваемую по умолчанию, а специально созданную на одном из разделов жесткого диска, например «**X:\www-srvs\**». Здесь и далее «**X**» — имя тома внешнего накопителя компьютера, на котором будет создана папка для размещения серверного программного обеспечения и программных кодов скриптов.



Рис. 3. Выбор целевой папки для установки сервера

Создание специальной папки упрощает перенос программного обеспечения сервера, что полезно, если разработка приложений будет отлаживаться на другом ПК. Создание симметричной файловой структуры исключит необходимость дополнительной отладки при вводе в эксплуатацию нового программного кода на целевой системе и исключит проблемы безопасности сервера в момент такой отладки.

После выбора целевой папки установки нажмите кнопку «**OK**», а в следующем окне программы установки кнопку «**Install**». В случае успешной установки и запуска в области «**Notification Area**» (уведомлений) панели «**Taskbar**» (задач) появится значок запущенного сервиса Apache в виде пера павлина с зеленым треугольником обозначающим выполняемое приложение.

Запустите браузер и в строке адреса наберите «**localhost**». Если на экране вы увидите то, что изображено на рис.4, то значит, установка прошла успешно и осталось провести тонкую настройку web сервера.



Рис. 4. Стартовое окно после установки сервера Apache

**Настройка Web-сервера Apache**

Настройка конфигурационных файлов Web-сервера – самый ответственный шаг при его установке. Сервер перечитывает конфигурационные файлы при запуске, а также при получении сигнала –HUP (жесткий рестарт) или -uSR1 (мягкий рестарт). Если сервер находится в рабочем состоянии, то при изменении конфигурации его рекомендуется перезапустить.

Одним из наиболее важных файлов конфигурации сервера Apache является файл «***httpd.conf***». В «***httpd.conf***»содержатся директивы, описывающие основные настройки Web-сервера. Файл представляет собой текстовый документ определенной семантики, что позволяет вносить в него изменения с помощью простейшего текстового редактора.

Однако далеко не всегда удобно вносить изменения настроек в текстовом виде. В качестве вспомогательных инструментов, упрощающих работу начинающим пользователям, программистами созданы специализированные приложения по управлению контентом конфигурационного файла «***httpd-conf***». Одним из таких приложений является программа «**ApacheConf**», распространяемая по принципу Shareware и имеющая возможность 30-ти дневного ознакомительного периода перед покупкой лицензии (рис.5).



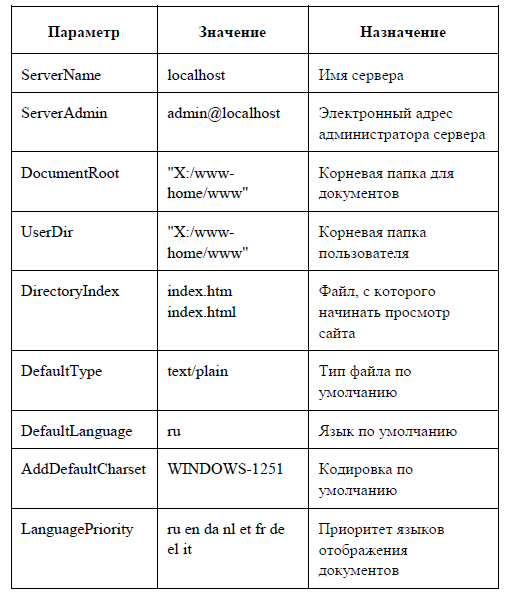
Рис***. 5.***Окно приложения ***ApacheConf***

Если приложения для визуального редактирования файла конфигурации нет, то изменение настроек вполне можно выполнить и вручную. Для этого необходимо внести в файл соответствующие изменения.

Прежде всего, необходимо определиться, в каком каталоге будут храниться документы. По умолчанию это «**/apache/share/htdocs**», однако это достаточно неудобно. В нашем случае предпочтительней разместить документы в папке «**X:\www-home\**» Здесь и далее «**X**» —имя тома внешнего накопителя компьютера, на котором будет создана папка для размещения серверного программного обеспечения и программных кодов скриптов.

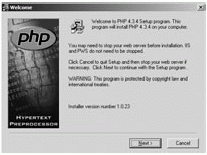
Кроме того, для корректной работы web-сервера с русскоязычными документами, необходимо внести соответствующие изменения в региональные переменные файла httpd.conf. Ниже приведены параметры настройки и возможные их значения, приводящие к корректному функционированию сервера на платформе Windows.

Таблица 1. Основные параметры первоначальной настройки ***Apache.***



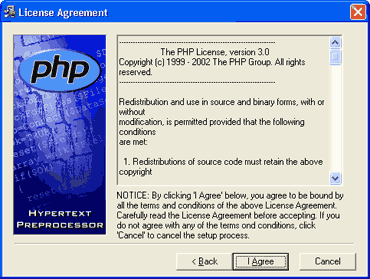
#### Установка PHP 4.3.4 под Windows

Перейдем к *установке* *PHP*. Скачать его *дистрибутив* можно с официального сайта *PHP* – [http://www.php.net](http://www.php.net/). Для удобства лучше выбрать автоматический инсталлятор, как и в случае с *сервером*. Самое первое окошко при такой *установке* *PHP* содержит приветствие и предупреждение о существовании авторских прав на этот продукт ([рис. 1.9](http://www.intuit.ru/studies/courses/42/42/lecture/27175?page=3#image.1.9)).



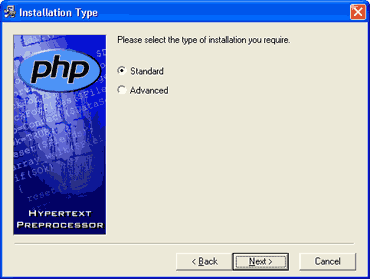
**Рис. 1.9.** Программа автоматической установки PHP

Далее нужно прочитать и согласиться (или не согласиться) с лицензией ([рис. 1.10](http://www.intuit.ru/studies/courses/42/42/lecture/27175?page=3#image.1.10)).



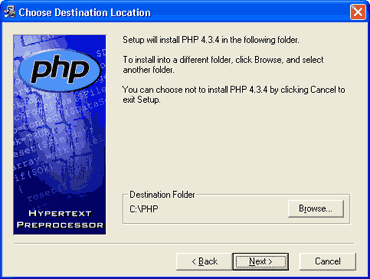
**Рис. 1.10.** Лицензионное соглашение

Затем выбираем тип *установки*: стандартный или для специалистов. Мы выберем стандартную *установку* ([рис. 1.11](http://www.intuit.ru/studies/courses/42/42/lecture/27175?page=3#image.1.11)).



**Рис. 1.11.** Тип установки

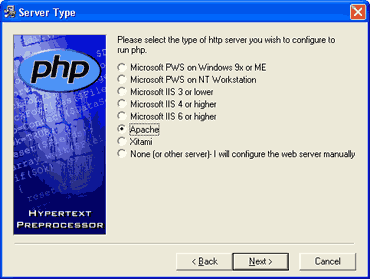
В следующем окне *программа* предложит выбрать директорию для *установки* *PHP*. Туда будут скопированы файлы библиотек, расширений, *интерпретатор* командной строки, модули и т.п. ([рис. 1.12](http://www.intuit.ru/studies/courses/42/42/lecture/27175?page=3#image.1.12)).



**Рис. 1.12.** Выбор папки, в которую будет установлен PHP

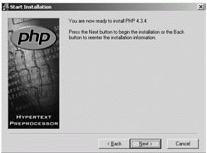
В следующем окошке предлагается ввести адрес SMTP- *сервера*, т.е. *сервера* отправки почтовых сообщений, и обратный адрес отправителя почты. Это окошко мы приводить не будем.

Далее предлагается выбрать из списка *сервер*, с которым будет работать *PHP*. Поскольку мы *установили* *Apache*, то выберем из списка именно его ([рис. 1.13](http://www.intuit.ru/studies/courses/42/42/lecture/27175?page=3#image.1.13)).



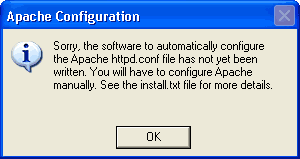
**Рис. 1.13.** Выбор сервера, на котором будет работать PHP

Следующее окошко начинает инсталляцию ([рис. 1.14](http://www.intuit.ru/studies/courses/42/42/lecture/27175?page=3#image.1.14)).



**Рис. 1.14.** Начало автоматической установки PHP

После копирования необходимых файлов мы получили следующее сообщение (см. [рис. 1.15](http://www.intuit.ru/studies/courses/42/42/lecture/27175?page=3#image.1.15)):



**Рис. 1.15.** Сообщение о невозможности автоматически настроить Apache для работы с PHP

Это значит, что нужно настраивать *сервер Apache* для работы с *PHP* вручную.

Сначала следует выбрать, как мы хотим *установить* *PHP*, поскольку он поставляется в двух видах: *CGI-скрипт* ( php.exe ) или набор *SAPI-модулей* (например, php4isapi.dll ), используемых *сервером*. Последний вариант обладает новыми возможностями, однако из-за недостаточной проработанности может функционировать недостаточно надежно, особенно на платформах старше Windows 2000 (может появляться куча ошибок 500, могут возникать сбои в других *серверных* модулях, таких как ASP). Так что, если нужна абсолютная стабильность – надо выбирать *установку* *PHP* в виде *CGI* выполнимого приложения.

Если мы хотим *установить* *PHP* как серверный модуль, то в файле конфигурации *сервера* ( *httpd.conf* ) нужно написать:

LoadModule php4\_module

c:/php/sapi/php4apache.dll

AddType application/x-httpd-php .php .phtml

AddModule mod\_php4.c

Если мы устанавливаем *PHP* как *cgi* - *программу*, то в *httpd.conf* нужно написать:

ScriptAlias /php/ "c:/php/"

AddType application/x-httpd-php .php .phtml

Action application/x-httpd-php "/php/php.exe"

В этом случае могут возникнуть проблемы с безопасностью. Рекомендуется исправить директорию, где лежит исполняемый файл *интерпретатора* ( c:\php\ ), на что-нибудь менее очевидное (например, на c:\abc\_php\ ). Мы советуем устанавливать *PHP* как серверный модуль.

Еще нужно отредактировать файл *php.ini* (в папке c:\Windows ), чтобы *PHP* "знал", где находится корневая директория *сервера*, где пользовательские директории, а где его собственные библиотеки расширений. За это в файле *php.ini* отвечают соответственно переменные ***doc\_root*** , ***user\_dir*** и ***extension\_dir*** . Зададим их таким образом:

doc\_root = "c:\Program Files\Apache

Group\Apache\htdocs"

user\_dir = "c:\users"

extension\_dir = "c:\php\extensions"

Кроме того, можно выбрать расширения, которые будут загружаться при запуске *PHP*. В реализацию *PHP* под Windows изначально входит очень мало расширений. Чтобы загрузить расширение, нужно раскомментировать в *php.ini* соответствующую ему строчку 'extension=php\_\*.dll'. Например, чтобы загрузить расширение для работы с MSSQL, нужно раскомментировать строку 'extension=php\_mssql.dll'. Некоторые расширения требуют дополнительных библиотек. Поэтому рекомендуется скопировать дополнительные библиотеки в системную директорию (из папки c:\php\dlls ). При первой *установке* следует настроить и протестировать *PHP* без расширений.

Для того чтобы *настройки*, выполненные в конфигурационных файлах *сервера* и *PHP* вступили в силу, нужно перезапустить *сервер*.

Проверим, работает ли *PHP*. Для этого создадим тестовый файл ( 1.php ) в директории пользователя ( c:\users\nina ) со следующим содержанием:

<?php

echo"<h1>Привет всем!</h1>";

?>

Запустим этот файл через браузер, набрав <http://localhost/~nina/1.php>. Если что-то не так, то на экран будет выведен текст этого файла. Если все хорошо, то наш *скрипт* должен обработаться *сервером* и вывести большими буквами строку "Привет всем!".

#### Дистрибутив Денвер

Мы достаточно подробно рассмотрели *установку* и *настройку* *PHP* для платформ Linux и Windows. Для тех, кто не желает вникать в устройство *PHP* и требуемых для его работы компонентов, существуют готовые дистрибутивы, содержащие все наиболее распространенные расширения *PHP*. Один из самых известных – *дистрибутив* ***Денвер*** (<http://dklab.ru/chicken/web/>). Инструкции по его *установке* можно прочитать на сайте разработчиков. Отметим только, что его *установка* достаточно проста и не требует особых навыков. Этот *дистрибутив* можно рекомендовать для начинающих программистов, желающих научиться работать с *PHP*. Для более серьезных задач лучше все же *установить* и настроить *PHP* самостоятельно.

**4.2 Основы синтаксиса PHP. Функции и области видимости**

Мы приступаем к изучению *основных элементов синтаксиса* языка *PHP*. Рассмотрим способы *разделения инструкций* и создания *комментариев*, *переменные*, *константы*, типы данных и *операторы*.

В качестве примера решим задачу *создания заготовки электронного письма*. Ее смысл заключается в следующем.

Допустим, у вас есть какое-то объявление и несколько разных людей, которым нужно это объявление отправить. Для этого вы делаете заготовку с содержанием объявления, внутри которого есть ряд изменяющихся (в зависимости от потенциального получателя) параметров.

**Основной синтаксис**

Первое, что нужно знать относительно синтаксиса *PHP*, – это то, как он встраивается в HTML-код, как *интерпретатор* узнает, что это код на языке *PHP*. В предыдущей лекции мы уже говорили об этом. Повторяться не будем, отметим только, что в примерах мы чаще всего будем использовать вариант <?php ?>, и иногда сокращенный вариант <? ?>.

**Разделение инструкций**

*Программа* на *PHP* (да и на любом другом языке программирования) – это набор команд (инструкций). Обработчику программы (*парсеру*) необходимо как-то отличать одну команду от другой. Для этого используются специальные символы – разделители. В *PHP* инструкции разделяются так же, как и в Cи или Perl, – каждое *выражение* заканчивается точкой с запятой.

Закрывающий *тег* " ?> " также подразумевает конец инструкции, поэтому перед ним точку с запятой не ставят. Например, следующие фрагменты кода эквивалентны:

<?php

echo "Hello, world!"; // точка с запятой

// в конце команды

// обязательна

?>

<?php

echo "Hello, world!" ?>

<!-- точка с запятой

опускается из-за "?>" -->

**Комментарии**

Часто при написании программ возникает необходимость делать какие-либо ***комментарии*** к коду, которые никак не влияют на сам код, а только поясняют его. Это важно при создании больших программ и в случае, если несколько человек работают над одной программой. При наличии *комментариев* в программе в ее коде разобраться гораздо проще. Кроме того, если решать задачу по частям, недоделанные части решения также удобно *комментировать*, чтобы не забыть о них в дальнейшем. Во всех языках программирования предусмотрена возможность включать *комментарии* в *код программы*. *PHP* поддерживает несколько видов *комментариев*: в стиле Cи, C++ и оболочки Unix. Символы // и # обозначают начало однострочных *комментариев*, /\* и \*/ – соответственно начало и конец многострочных *комментариев*.

<?php

echo "Меня зовут Вася";

// Это однострочный комментарий

// в стиле С++

echo "Фамилия моя Петров";

/\* Это многострочный комментарий.

Здесь можно написать несколько строк.

При исполнении программы все, что

находится здесь (закомментировано),

будет игнорировано. \*/

echo "Я изучаю PHP в INTUIT.ru";

# Это комментарий в стиле

# оболочки Unix

?>

Пример 2.1. Использование комментариев в PHP

### Переменные, константы и операторы

Важным элементом каждого языка являются *переменные*, *константы* и *операторы*, применяемые к этим *переменным* и *константам*. Рассмотрим, как выделяются и обрабатываются эти элементы в *PHP*.

#### Переменные

*Переменная* в PHP обозначается знаком доллара, за которым следует ее имя. Например:

$my\_var

Имя *переменной* *чувствительно к регистру*, т.е. *переменные* $my\_var и $My\_var различны.

Имена *переменных* соответствуют тем же правилам, что и остальные наименования в PHP: правильное имя *переменной* должно начинаться с буквы или символа подчеркивания с последующими в любом количестве буквами, цифрами или символами подчеркивания.

В PHP 3 *переменные* всегда присваивались по значению. То есть когда вы присваиваете выражение *переменной*, все значения оригинального выражения копируются в эту *переменную*. Это означает, к примеру, что после присвоения одной *переменной* значения другой, изменение одной из них не влияет на значение другой.

<?php

$first = ' Text '; // Присваиваем $first

// значение

// ' Text '

$second = $first; // Присваиваем $second

// значение

// переменной $first

$first = ' New text '; // Изменяем

// значение

// $first

// на ' New text '

echo "Переменная с именем first равна " .$first <br>";

// выводим значение $first

echo "Переменная с именем second равна " .$second";

// выводим значение $second

?>

Пример 2.2. Присваивание по значению

Результат работы этого скрипта будет следующим:

Переменная с именем first равна New text

Переменная с именем second равна Text

PHP 4, кроме этого, предлагает еще один способ присвоения значений *переменным*: присвоение по ссылке. Для того, чтобы присвоить значение *переменной* по ссылке, это значение должно иметь имя, т.е. оно должно быть представлено какой-либо *переменной*. Чтобы указать, что значение одной *переменной* присваивается другой *переменной* по ссылке, нужно перед именем первой *переменной* поставить знак амперсанд &.

Рассмотрим тот же пример, что и выше, только будем присваивать значение *переменной* first *переменной* second по ссылке:

<?php

$first = ' Text '; // Присваиваем $first

// значение ' Text '

$second = &$first;

/\* Делаем ссылку на $first через $second.

Теперь значения этих переменных

будут всегда совпадать \*/

// Изменим значение $first

// на ' New text '

$first = ' New text ';

echo "Переменная с именем first " .

"равна .$first <br>";

// выведем значения обеих переменных

echo "Переменная с именем second " .

"равна $second";

?>

Пример 2.3. Присваивание по ссылке

Этот скрипт выведет следующее:

Переменная с именем first равна New text.

Переменная с именем second равна New text.

То есть вместе с *переменной* $first изменилась и *переменная* $second.

#### Константы

Для хранения постоянных величин, т.е. таких величин, значение которых не меняется в ходе выполнения скрипта, используются ***константы***. Такими величинами могут быть математические *константы*, пароли, пути к файлам и т.п. Основное отличие *константы* от *переменной* состоит в том, что ей нельзя присвоить значение больше одного раза и ее значение нельзя аннулировать после ее объявления. Кроме того, у *константы* нет приставки в виде знака доллара и ее нельзя определить простым присваиванием значения. Как же тогда можно определить *константу*? Для этого существует специальная функция define() . Ее синтаксис таков:

define("Имя\_константы",

"Значение\_константы",

[Нечувствительность\_к\_регистру])

По умолчанию имена *констант* *чувствительны к регистру*. Для каждой *константы* это можно изменить, указав в качестве значения аргумента Нечувствительность\_к\_регистру значение True. Существует соглашение, по которому имена *констант* всегда пишутся в верхнем регистре.

Получить значение *константы* можно, указав ее имя. В отличие от *переменных*, не нужно предварять имя *константы* символом $. Кроме того, для получения значения *константы* можно использовать функцию *constant*() с именем *константы* в качестве параметра.

<?php

// определяем константу

// PASSWORD

define("PASSWORD","qwerty");

// определяем регистронезависимую

// константу PI со значением 3.14

define("PI","3.14", True);

// выведем значение константы PASSWORD,

// т.е. qwerty

echo (PASSWORD);

// тоже выведет qwerty

echo constant("PASSWORD");

echo (password);

/\* выведет password и предупреждение,

поскольку мы ввели регистрозависимую

константу PASSWORD \*/

echo pi;

// выведет 3.14, поскольку константа PI

// регистронезависима по определению

?>

Пример 2.4. Константы в PHP

Кроме *констант*, объявляемых пользователем, о которых мы только что рассказали, в PHP существует ряд *констант*, определяемых самим интерпретатором. Например, *константа* \_\_FILE\_\_ хранит имя файла программы (и путь к нему), которая выполняется в данный момент, \_\_FUNCTION\_\_ содержит имя функции, \_\_CLASS\_\_ – имя класса, PHP\_VERSION – версия интерпретатора PHP. Полный список предопределенных *констант* можно получить, прочитав руководство по PHP.

#### Операторы

***Операторы*** позволяют выполнять различные действия с *переменными*, *константами* и выражениями. Мы еще не упоминали о том, что такое выражение. Выражение можно определить как все, что угодно, что имеет значение. *Переменные* и *константы* – это основные и наиболее простые формы выражений. Существует множество операций (и соответствующих им *операторов* ), которые можно производить с выражениями. Рассмотрим некоторые из них подробнее.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 2.1. *Арифметические операторы* | | | |
| **Обозначение** | | **Название** | **Пример** |
| + | | Сложение | $a + $b |
| - | | Вычитание | $a - $b |
| \* | | Умножение | $a \* $b |
| / | | Деление | $a / $b |
| % | | Остаток от деления | $a % $b |
| Таблица 2.2. Строковые операторы | | | | | |
| **Обозначение** | **Название** | | | | **Пример** |
| . | *Конкатенация* ( *сложение строк* ) | | | | $c = $a . $b  (это строка, состоящая из $a и $b ) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 2.3. Операторы присваивания | | | | | |
| **Обозначение** | **Название** | **Описание** | | **Пример** | |
| = | Присваивание | *Переменной* слева от *оператора* будет присвоено значение, полученное в результате выполнения каких-либо операций или *переменной* / *константы* с правой стороны | | $a = ($b = 4) +5;  ( $a будет равна 9, $b будет равна 4 ) | |
| += |  | Сокращение. Прибавляет к *переменной* число и затем присваивает ей полученное значение | | $a += 5;  (эквивалентно $a = $a + 5; ) | |
| .= |  | Сокращенно обозначает комбинацию операций *конкатенации* и присваивания (сначала добавляется строка, потом полученная строка записывается в *переменную* ) | | $b = "Привет ";  $b .= "всем";  (эквивалентно $b = $b . "всем"; )  В результате: $b="Привет всем" | |
| Таблица 2.4. *Логические операторы* | | | | | |
|  | | | | | |
| **Обозначение** | **Название** | | **Описание** | | **Пример** |
| and | И | | $a и $b истинны ( True ) | | $a and $b |
| && | И | |  | | $a && $b |
| or | Или | | Хотя бы одна из *переменных* $a или $b истинна (возможно, что и обе) | | $a or $b |
| || | Или | |  | | $a || $b |
| xor | Исключающее или | | Одна из *переменных* истинна. Случай, когда они обе истинны, исключается | | $a xor $b |
| ! | Инверсия (NOT) | | Если $a=True, то !$a=False и наоборот | | ! $a |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 2.5. Операторы сравнения | | | | | |
| **Обозначение** | **Название** | | **Описание** | | **Пример** |
| == | Равенство | | Значения *переменных* равны | | $a == $b |
| === | Эквивалентность | | Равны значения и *типы* *переменных* | | $a === $b |
| != | Неравенство | | Значения *переменных* не равны | | $a != $b |
| <> | Неравенство | |  | | $a <> $b |
| !== | Неэквивалентность | | *Переменные* не эквивалентны | | $a !== $b |
| < | Меньше | |  | | $a < $b |
| > | Больше | |  | | $a > $b |
| <= | Меньше или равно | |  | | $a <= $b |
| >= | Больше или равно | |  | | $a >= $b |
| Таблица 2.6. Операторы инкремента и декремента | | | | | | |
| **Обозначение** | **Название** | **Описание** | | **Пример** | | |
| ++$a | Пре- *инкремент* | Увеличивает $a на единицу и возвращает $a | | <?  $a=4;  echo "Должно быть 4:" .$a++;  echo "Должно быть 5:" .++$a;  ?> | | |
| $a++ | Пост- *инкремент* | Возвращает $a, затем увеличивает $a на единицу | |  | | |
| --$a | Пре- *декремент* | Уменьшает $a на единицу и возвращает $a | |  | | |
| $a-- | Пост- *декремент* | Возвращает $a, затем уменьшает $a на единицу | |  | | |

### Типы данных

*PHP* поддерживает восемь простых *типов данных*.

Четыре скалярных *типа*:

* *boolean (логический)* ;
* *integer (целый)* ;
* *float (с плавающей точкой)* ;
* *string (строковый)*.

Два смешанных *типа*:

* *array (массив)* ;
* *object (объект)*.

И два специальных *типа*:

* *resource (ресурс)* ;
* *NULL*.

В *PHP* не принято явное объявление *типов* *переменных*. Предпочтительнее, чтобы это делал сам *интерпретатор* во *время выполнения* программы в зависимости от контекста, в котором используется *переменная*. Рассмотрим по порядку все перечисленные *типы данных*.

#### Тип boolean (булев или логический тип)

Этот простейший *тип* выражает истинность значения, то есть *переменная* этого *типа* может иметь только два значения – истина TRUE или ложь FALSE .

Чтобы определить булев *тип*, используют ключевое слово TRUE или FALSE. Оба регистронезависимы.

<?php

$test = True;

?>

Пример 2.5. Логический тип

Логические *переменные* используются в различных *управляющих конструкциях* (циклах, условиях и т.п., более подробно речь о них пойдет в одной из следующих лекций). Иметь *логический тип*, т.е. принимать только два значения, истину или ложь, могут также и некоторые *операторы* (например, *оператор* равенства). Они также используются в *управляющих конструкциях* для проверки каких-либо условий. Например, в условной конструкции проверяется истинность значения *оператора* или *переменной* и в зависимости от результата проверки выполняются те или иные действия. Здесь условие может быть истинно или ложно, что как раз и отражает *переменная* и *оператор**логического типа*.

<?php

// Оператор '==' проверяет равенство

// и возвращает

// булево значение

if ($know == False) { // если $know

// имеет значение

// false

echo "Изучай PHP!";

}

if (!$know) { // то же самое, что

// и выше, т.е. проверка

// имеет ли $know значение

// false

echo "Изучай PHP!";

}

/\* оператор == проверяет, совпадает ли

значение переменной $know со строкой

"Изучай PHP". Если совпадает, то

возвращает true, иначе – false.

Если возвращено true, то выполняется

то, что внутри фигурных скобок \*/

if ($know == "Изучай PHP")

{ echo "Начал изучать"; }

?>

Пример 2.6. Использование логического типа

#### Тип integer (целые)

Этот *тип* задает число из множества целых чисел Z = {..., -2, -1, 0, 1, 2, ...}. Целые могут быть указаны в десятичной, шестнадцатеричной или *восьмеричной системе счисления*, по желанию с предшествующим знаком " - " или " + ".

Если вы используете восьмеричную систему счисления, вы должны предварить число 0 (нулем), для использования шестнадцатеричной системы нужно поставить перед числом 0x.

<?php

# десятичное число

$a = 1234;

# отрицательное число

$a = -123;

# восьмеричное число (эквивалентно

# 83 в десятичной системе)

$a = 0123;

# шестнадцатеричное число (эквивалентно

# 26 в десятичной системе)

$a = 0x1A;

?>

Размер *целого* зависит от платформы, хотя, как правило, максимальное значение около двух миллиардов (это 32-битное знаковое). Беззнаковые *целые* PHP не поддерживает.

Если вы определите число, превышающее пределы *целого типа*, оно будет интерпретировано как *число с плавающей точкой*. Также если вы используете *оператор*, результатом работы которого будет число, превышающее пределы *целого*, вместо него будет возвращено *число с плавающей точкой*.

В PHP не существует *оператора* деления *целых*. Результатом 1/2 будет *число с плавающей точкой* 0.5. Вы можете привести значение к *целому*, что всегда округляет его в меньшую сторону, либо использовать функцию round(), округляющую значение по стандартным правилам. Для преобразования *переменной* к конкретному *типу* нужно перед *переменной* указать в скобках нужный *тип*. Например, для преобразования *переменной* $a=0.5 к *целому типу* необходимо написать (integer)(0.5) или (integer) $a или использовать сокращенную запись (int)(0.5). Возможность явного приведения *типов* по такому принципу существует для всех *типов данных* (конечно, не всегда значение одного *типа* можно перевести в другой *тип* ). Мы не будем углубляться во все тонкости приведения *типов*, поскольку PHP делает это автоматически в зависимости от контекста.

#### Тип float (числа с плавающей точкой)

*Числа с плавающей точкой* (они же числа двойной точности или *действительные числа*) могут быть определены при помощи любого из следующих синтаксисов:

<?php

$a = 1.234;

$b = 1.2e3;

$c = 7E-10;

?>

Размер *числа с плавающей точкой* зависит от платформы, хотя максимум, как правило, ~1.8e308 с точностью около 14 десятичных цифр.

#### Тип object (объекты)

***Объекты*** – *тип данных*, пришедший из объектно-ориентированного программирования (*ООП*). Согласно принципам *ООП*, класс – это набор *объектов*, обладающих определенными свойствами и методами работы с ним, а *объект* соответственно – экземпляр класса. Например, программисты – это класс людей, которые пишут программы, изучают компьютерную литературу и, кроме того, как все люди, имеют имя и фамилию. Теперь, если взять одного конкретного программиста, Васю Иванова, то можно сказать, что он является *объектом* класса программистов, обладает теми же свойствами, что и другие программисты, тоже имеет имя, пишет программы и т.п.

В PHP для доступа к методам *объекта* используется *оператор* ->. Для инициализации *объекта* используется выражение new, создающее в *переменной* экземпляр *объекта*.

<?php

//создаем класс людей

class Person

{

// метод, который обучает человека PHP

function know\_php()

{

echo "Теперь я знаю PHP";

}

}

$bob = new Person; // создаем объект

// класса человек

$bob -> know\_php(); // обучаем его PHP

?>

Пример 2.12. Объекты в PHP

Более подробно реализацию принципов *ООП* в языке PHP мы рассмотрим в одной из следующих лекций.

#### Тип resource (ресурсы)

***Ресурс*** – это специальная *переменная*, содержащая ссылку на внешний *ресурс* (например, соединение с базой данных). *Ресурсы* создаются и используются специальными функциями (например, *mysql\_connect*(), pdf\_new() и т.п.).

#### Тип Null

Специальное значение *NULL* говорит о том, что *переменная* не имеет значения.

*Переменная* считается *NULL*, если:

* ей была присвоена *константа* *NULL* ( $var = NULL );
* ей еще не было присвоено какое-либо значение;
* она была удалена с помощью *unset* ().

Существует только одно значение *типа* *NULL* – регистронезависимое ключевое слово NULL.

### Условные операторы

#### Оператор if

Это один из самых важных операторов многих языков, включая PHP. Он позволяет выполнять фрагменты кода в зависимости от условия. Структуру оператора *if* можно представить следующим образом:

if (выражение) блок\_выполнения

Здесь выражение есть любое правильное PHP-выражение (т.е. все, что имеет значение). В процессе обработки скрипта выражение преобразуется к *логическому типу*. Если в результате преобразования значение выражения истинно ( True ), то выполняется блок\_выполнения. В противном случае блок\_выполнения игнорируется. Если блок\_выполнения содержит несколько команд, то он должен быть заключен в фигурные скобки { }.

Правила преобразования выражения к *логическому типу*:

1. В FALSE преобразуются следующие значения:
   * логическое False
   * целый ноль ( 0 )
   * действительный ноль ( 0.0 )
   * пустая строка и строка "0"
   * массив без элементов
   * объект без переменных (подробно об объектах будет рассказано в одной из следующих лекций)
   * специальный тип NULL
2. Все остальные значения преобразуются в TRUE.

<?

$names = array("Иван","Петр","Семен");

if ($names[0]=="Иван") {

echo "Привет, Ваня!";

$num = 1;

$account = 2000;

}

if ($num) echo "Иван первый в списке!";

$bax = 30;

if ($account > 100\*$bax+3)

echo "Эта строчка не появится

на экране, так как условие не выполнено";

?>

Пример 3.1. Условный оператор if

#### Оператор else

Мы рассмотрели только одну, основную часть оператора *if* . Существует несколько расширений этого *оператора. Оператор* *else*расширяет *if* на случай, если проверяемое в *if* выражение является неверным, и позволяет выполнить какие-либо действия при таких условиях.

Структуру оператора *if* , расширенного с помощью оператора *else* , можно представить следующим образом:

if (выражение) блок\_выполнения

else блок\_выполнения1

Эту конструкцию if...else можно интерпретировать примерно так: если выполнено условие (т.е. выражение=true ), то выполняем действия из блока\_выполнения, иначе – действия из блока\_выполнения1. Использовать оператор *else* не обязательно.

Посмотрим, как можно изменить предыдущий пример, учитывая необходимость совершения действий и в случае невыполнения условия.

<?

$names = array("Иван","Петр","Семен");

if ($names[0]=="Иван") {

echo "Привет, Ваня!";

$num = 1;

$account = 2000;

} else {

echo "Привет, $names[0].

А мы ждали Ваню :(";

}

if ($num) echo "Иван первый в списке!";

else echo "Иван НЕ первый в списке?!";

$bax = 30;

if ($account > 100\*$bax+3)

echo "Эта строка не появится на экране,

так как условие не выполнено";

else echo "Зато появится эта строка!";

?>

Пример 3.2. Оператор else

#### Оператор elseif

Еще один способ расширения *условного оператора* *if* – использование оператора *elseif* . *elseif* – это комбинация *else* и *if* . Как и *else* , он расширяет *if* для выполнения различных действий в том случае, если условие, проверяемое в *if* , неверно. Но в отличие от *else* , альтернативные действия будут выполнены, только если *elseif*-условие является верным. Структуру оператора *if* , расширенного с помощью операторов *else* и *elseif* , можно представить следующим образом:

if (выражение) блок\_выполнения

elseif(выражение1) блок\_выполнения1

...

else блок\_выполненияN

Операторов *elseif* может быть сразу несколько в одном if-блоке. *Elseif*-утверждение будет выполнено, только если предшествующее if-условие является False, все предшествующие *elseif*-условия являются False, а данное *elseif*-условие – True.

<?

$names = array("Иван","Петр","Семен");

if ($names[0]=="Иван") {

// если первое имя в массиве Иван

echo "Привет, Ваня!";

}elseif ($names[0] == "Петр"){

// если первое имя

// не Иван, а Петр

echo "Привет, Петя!";

}elseif ($names[0] == "Семен"){

// если первое имя не

// Иван, не Петр, а Семен

echo "Привет, Сеня!";

}else {

// если первое имя не Иван,

// не Петр и не Семен

echo "Привет, $names[0]. А ты кто такой?";

}

?>

Пример 3.3. Оператор elseif

#### Оператор switch

Еще одна конструкция, позволяющая проверять условие и выполнять в зависимости от этого различные действия, – это *switch* . На русский язык название данного оператора можно перевести как "переключатель". И смысл у него именно такой. В зависимости от того, какое значение имеет переменная, он переключается между различными блоками действия. *switch* очень похож на оператор if...*elseif*...else или набор операторов *if* . Структуру *switch* можно записать следующим образом:

switch (выражение или переменная){

case значение1:

блок\_действий1

break;

case значение2:

блок\_действий2

break;

...

default:

блок\_действий\_по\_умолчанию

}

В отличие от *if* , здесь значение выражения не приводится к логическому типу, а просто сравнивается со значениями, перечисленными после ключевых слов *case* ( значение1, значение2 и т.д.). Если значение выражения совпало с каким-то вариантом, то выполняется соответствующий блок\_действий – от двоеточия после совпавшего значения до конца *switch* или до первого оператора *break* , если таковой найдется. Если значение выражения не совпало ни с одним из вариантов, то выполняются действия по умолчанию ( блок\_действий\_по\_умолчанию ), находящиеся после ключевого слова default. Выражение в *switch* вычисляется только один раз, а в операторе *elseif* – каждый раз, поэтому, если выражение достаточно сложное, то *switch* работает быстрее.

Пример [3.3](http://www.intuit.ru/studies/courses/42/42/lecture/27179?page=1#example.3.3) можно переписать с использованием *switch* следующим образом:

<?

$names = array("Иван","Петр","Семен");

switch ($names[0]){

case "Иван":

echo "Привет, Ваня!";

break;

case "Петр":

echo "Привет, Петя!";

break;

case "Семен":

echo "Привет, Сеня!";

break;

default:

echo "Привет, $names[0].

А как Вас зовут?";

}

?>

Если в этом примере опустить оператор *break* , например, в *case* "Петр":, то, если переменная окажется равной строке "Петр", после вывода на экран сообщения "Привет, Петя!" программа пойдет дальше и выведет также сообщение "Привет, Сеня!" и только потом, встретив *break* , продолжит свое выполнение за пределами *switch* .

Для конструкции *switch* , как и для *if* , возможен *альтернативный синтаксис*, где открывающая *switch* фигурная скобка заменяется двоеточием, а закрывающая – endswitch; соответственно.

### Циклы

В *PHP* существует несколько конструкций, позволяющих выполнять повторяющиеся действия в зависимости от условия. Это циклы *while* , *do..while* , *foreach* и *for* . Рассмотрим их более подробно.

#### while

Структура:

while (выражение) { блок\_выполнения }

либо

while (выражение): блок\_выполнения endwhile;

*while* – простой цикл. Он предписывает PHP выполнять команды блока\_выполнения до тех пор, пока выражение вычисляется как True(здесь, как и в *if* , происходит приведение выражения к логическому типу). Значение выражения проверяется каждый раз в начале цикла, так что, даже если его значение изменилось в процессе выполнения блока\_выполнения, цикл не будет остановлен до конца итерации (т.е. пока все команды блока\_выполнения не будут исполнены).

<?

//эта программа напечатает все четные цифры

$i = 1;

while ($i < 10) {

if ($i % 2 == 0) print $i;

// печатаем цифру, если она четная

$i++;

// и увеличиваем $i на единицу

}

?>

Пример 3.5. Оператор while

#### do... while

Циклы *do..while* очень похожи на циклы *while* , с той лишь разницей, что истинность выражения проверяется в конце цикла, а не в начале. Благодаря этому блок\_выполнения цикла do...while гарантированно выполняется хотя бы один раз.

Структура:

do {блок\_выполнения} while (выражение);

<?

// эта программа напечатает число 12, несмотря на то

// что условие цикла не выполнено

$i = 12;

do{

if ($i % 2 == 0) print $i;

// если число четное, то печатаем его

$i++;

// увеличиваем число на единицу

}while ($i<10)

?>

Пример 3.6. Оператор do..while

#### for

Это самые сложные циклы в PHP. Они напоминают соответствующие циклы C.

Структура:

for (выражение1; выражение2; выражение3) {блок\_выполнения}

либо

for (выражение1; выражение2; выражение3): блок\_выполнения endfor;

Здесь, как мы видим, условие состоит сразу из трех выражений. Первое выражение выражение1 вычисляется безусловно один раз в начале цикла. В начале каждой итерации вычисляется выражение2. Если оно является True, то цикл продолжается и выполняются все команды блока\_выполнения. Если выражение2 вычисляется как False, то исполнение цикла останавливается. В конце каждой итерации (т.е. после выполнения всех команд блока\_выполнения ) вычисляется выражение3.

Каждое из выражений 1, 2, 3 может быть пустым. Если выражение2 является пустым, то это значит, что цикл должен выполняться неопределенное время (в этом случае PHP считает это выражение всегда истинным). Это не так бесполезно, как кажется, ведь цикл можно останавливать, используя оператор *break* .

Например, все четные цифры можно вывести с использованием цикла *for* таким образом:

<?php

for ($i=0; $i<10; $i++){

if ($i % 2 == 0) print $i;

// печатаем четные числа

}

?>

Если опустить второе выражение (условие $i<10 ), то такую же задачу можно решить, останавливая цикл оператором *break* .

<?php

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

if ($i>=10) break;

// если $i больше или равно 10,

// то прекращаем работу цикла

if ($i % 2 == 0) print $i;

// если число четное,

// то печатаем его

}

?>

Можно опустить все три выражения. В этом случае просто не будет задано начальное значение счетчика $i и оно не будет изменяться каждый раз в конце цикла. Все эти действия можно записать в виде отдельных команд либо в блоке\_выполнения, либо перед циклом:

<?php

$i=2; // задаем начальное значение счетчика

for ( ; ; ){

if ($i>=10) break;

// если $i больше или равно 10,

// то прекращаем работу цикла

if ($i % 2 == 0) print $i;

// если число четное,

// то печатаем его

$i++; // увеличиваем счетчик на единицу

}

?>

В третье выражение конструкции *for* можно записывать через запятую сразу несколько простейших команд. Например, если мы хотим просто вывести все цифры, то программу можно записать совсем просто:

<?php

for ($i=0; $i<10; print $i, $i++)

/\* Если блок\_выполнения не содержит команд

или содержит только одну команду,

фигурные скобки, в которые он заключен,

можно опускать\*/

?>

#### foreach

Еще одна полезная конструкция. Она появилась только в *PHP4* и предназначена исключительно для работы с массивами.

Синтаксис:

foreach ($array as $value) {блок\_выполнения}

либо

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

{блок\_выполнения}

В первом случае формируется цикл по всем элементам массива, заданного переменной $array. На каждом шаге цикла значение текущего элемента массива записывается в переменную $value, и внутренний счетчик массива передвигается на единицу (так что на следующем шаге будет виден следующий элемент массива). Внутри блока\_выполнения значение текущего элемента массива может быть получено с помощью переменной $value. Выполнение блока\_выполнения происходит столько раз, сколько элементов в массиве $array.

Вторая форма записи в дополнение к перечисленному выше на каждом шаге цикла записывает ключ текущего элемента массива в переменную $key, которую тоже можно использовать в блоке\_выполнения.

Когда *foreach* начинает исполнение, внутренний указатель массива автоматически устанавливается на первый элемент.

<?php

$names = array("Иван","Петр","Семен");

foreach ($names as $val) {

echo "Привет, $val <br>";

// выведет всем приветствие

}

foreach ($names as $k => $val) {

// кроме приветствия,

// выведем номера в списке, т.е. ключи

echo "Привет, $val !

Ты в списке под номером $k <br>";

}

?>

Пример 3.7. Оператор foreach

### Операторы передачи управления

Иногда требуется немедленно завершить работу *цикла* либо отдельной его итерации. Для этого используют *операторы* *break* и *continue*.

#### Break

Оператор *break* заканчивает выполнение текущего цикла, будь то *for* , *foreach* , *while* , *do..while* или *switch* . *break*может использоваться с числовым аргументом, который говорит, работу скольких управляющих структур, содержащих его, нужно завершить.

<?php

$i=1;

while ($i) {

$n = rand(1,10);

// генерируем произвольное число

// от 1 до 10

echo "$i:$n ";

// выводим номер итерации и

// сгенерированное число

if ($n==5) break;

/\* Если было сгенерировано число 5,

то прекращаем работу цикла. В этом случае

все, что находится после этой строчки

внутри цикла, не будет выполнено \*/

echo "Цикл работает <br>";

$i++;

}

echo "<br>Число итераций цикла $i ";

?>

Пример 3.8. Оператор break

Результатом работы этого скрипта будет примерно следующее:

1:7 Цикл работает

2:2 Цикл работает

3:5

Число итераций цикла 3

Если после оператора *break* указать число, то прервется именно такое количество содержащих этот *оператор циклов*. В приведенном выше примере это неактуально, поскольку *вложенных циклов* нет. Немного изменим наш скрипт:

<?php

$i=1;

while ($i) {

$n = rand(1,10);

// генерируем произвольное число

// от 1 до 10

switch ($n){

case 5:

echo "<font color=blue>

Выход из switch (n=$n)</font>";

break 1;

// прекращаем работу switch

// (первого содержащего break цикла)

case 10:

echo "<font color=red>

Выход из switch и

while (n=$n)</font>";

break 2;

// прекращаем работу switch и while

// (двух содержащих break циклов)

default:

echo "switch работает (n=$n), ";

}

echo " while работает – шаг $i <br>";

$i++;

}

echo "<br>Число итераций цикла $i ";

?>

#### continue

Иногда нужно не полностью прекратить работу цикла, а только начать его новую итерацию. Оператор *continue* позволяет пропустить дальнейшие инструкции из блока\_выполнения любого цикла и продолжить выполнение с нового круга. *continue* можно использовать с числовым аргументом, который указывает, сколько содержащих его *управляющих конструкций* должны завершить работу.

Заменим в примере предыдущего параграфа оператор *break* на *continue* . Кроме того, ограничим число шагов цикла четырьмя.

<?php

$i=1;

while ($i<=4) {

$n = rand(1,10);

// генерируем произвольное число

// от 1 до 10

echo "$i:$n ";

// выводим номер итерации и

// сгенерированное число

if ($n==5) {

echo "Новая итерация <br>";

continue;

/\* Если было сгенерировано число 5,

то начинаем новую итерацию цикла,

$i не увеличивается \*/

}

echo "Цикл работает <br>";

$i++;

}

--$i;

echo "<br>Число итераций цикла $i ";

?>

Результатом работы этого скрипта будет

1:10 Цикл работает

2:5 Новая итерация

2:1 Цикл работает

3:1 Цикл работает

4:1 Цикл работает

Число итераций цикла 4

Заметим, что после выполнения оператора *continue* работа цикла не заканчивается. В примере счетчик цикла не меняется в случае получения числа 5, поскольку он находится после оператора *continue* . Фактически с помощью *continue* мы пытаемся избежать ситуации, когда будет сгенерировано число 5. Поэтому можно было просто написать, заменив оператор *continue* на проверку истинности выражения:

<?php

$i=1;

while ($i<4) {

$n = rand(1,10);

// генерируем произвольное число

// от 1 до 10

if ($n!==5) {

echo "$i:$n <br>";

// выводим номер итерации

// и сгенерированное число

$i++;

}

}

?>

В PHP существует одна особенность использования оператора *continue* – в конструкциях *switch* он работает так же, как и *break* . Если *switch* находится внутри цикла и нужно начать новую итерацию цикла, следует использовать continue 2.

### Решение задачи

Теперь вернемся к задаче, которую мы поставили в самом начале лекции. Напомним, что она состояла в *составлении письма* разным людям по поводу разных событий. Попытаемся использовать для решения этой задачи изученные средства – *переменные*, *операторы*, *константы*, *строки* и *массивы*. В зависимости от получателя изменяется событие и обращение, указанные в письме, поэтому естественно вынести эти величины в *переменные*. Более того, поскольку событий и людей много, удобно использовать *переменные* *типа* *массив*. Подпись в письме остается постоянной всегда, поэтому логично задать ее как *константу*. Чтобы не писать слишком длинные и громоздкие *строки*, используем *оператор* *конкатенации*. Итак, вот что получилось:

<?

// пусть наша подпись

// будет константой

define("SIGN","С уважением, Вася");

// зададим массивы людей и событий

$names = array("Иван Иванович",

"Петр Петрович",

"Семен Семенович");

$events = array(

"f" => "день открытых дверей",

"o" => "открытие выставки",

"p" => "бал выпускников");

// составим текст приглашения

$str = "Уважаемый (ая), $names[0].";

$str .= "<br>Приглашаем Вас на ".

$events["f"];

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

echo $str; // выведем текст на экран

?>

### Заключение

Итак, в этой лекции мы познакомились с *основами синтаксиса* языка *PHP*, научились работать с *переменными* различных *типов*, *константами*, *операторами*, познакомились со всеми существующими в *PHP* *типами данных*. Говоря о таких *типах данных*, как *строки* и *массивы*, мы разобрали только самые основы. Эти конструкции настолько удобны и просты в использовании, что заслуживают отдельного разговора. Подробности будут изложены в последующих лекциях. Решая задачу, мы старались использовать только имеющиеся знания и не забегать вперед, поэтому решение не слишком подходит для практического применения. В следующей лекции мы исправим этот недостаток и создадим более универсальный *шаблон* электронного письма.

### Основы клиент-серверных технологий

В самом начале курса мы уже говорили о том, что *PHP* – это скриптовый язык, обрабатываемый *сервером*. Сейчас мы хотим уточнить, что же такое *сервер*, какие функции он выполняет и какие вообще бывают *серверы*. Если речь идет о *сервере*, невольно всплывает в памяти понятие *клиента*. Все потому, что эти два понятия неразрывно связаны. Объединяет их компьютерная *архитектура клиент-сервер*. Обычно, когда говорят " *сервер* ", имеют в виду *сервер* в *архитектуре клиент-сервер*, а когда говорят " *клиент* " – имеют в виду *клиент* в этой же *архитектуре*. Так что же это за *архитектура*? Суть ее в том, чтобы разделить функции между двумя подсистемами: *клиентом*, который отправляет *запрос* на выполнение каких-либо действий, и *сервером*, который выполняет этот *запрос*. Взаимодействие между *клиентом* и *сервером* происходит посредством стандартных специальных протоколов, таких как *TCP/IP* и *z39.50*. На самом деле протоколов очень много, они различаются по уровням. Мы рассмотрим только протокол прикладного уровня *HTTP* (чуть позднее), поскольку для решения наших программистских задач нужен только он. А пока вернемся к *клиент-серверной архитектуре* и разберемся, что же такое *клиент* и что такое *сервер*.

*Сервер* представляет собой набор программ, которые контролируют выполнение различных процессов. Соответственно, этот набор программ установлен на каком-то компьютере. Часто компьютер, на котором установлен *сервер*, и называют *сервером*. Основная функция компьютера-сервера – по запросу *клиента* запустить какой-либо *определенный процесс* и отправить *клиенту* результаты его работы.

*Клиентом* называют любой процесс, который пользуется услугами *сервера*. *Клиентом* может быть как пользователь, так и программа. Основная задача *клиента* – выполнение приложения и осуществление связи с *сервером*, когда этого требует приложение. То есть *клиент*должен предоставлять пользователю интерфейс для работы с приложением, реализовывать логику его работы и при необходимости отправлять задания *серверу* .

Взаимодействие между *клиентом* и *сервером* начинается по инициативе *клиента*. *Клиент* запрашивает вид обслуживания, устанавливает *сеанс*, получает нужные ему результаты и сообщает об окончании работы.

Услугами одного *сервера* чаще всего пользуется несколько *клиентов* одновременно. Поэтому каждый *сервер* должен иметь достаточно большую *производительность* и обеспечивать *безопасность данных*.

Логичнее всего устанавливать *сервер* на компьютере, входящем в какую-либо *сеть*, локальную или глобальную. Однако можно устанавливать *сервер* и на отдельно стоящий *компьютер* (тогда он будет являться одновременно и *клиентом* и *сервером* ).

Существует множество типов *серверов*. Вот лишь некоторые из них.

* **Видеосервер**

Такой *сервер* специально приспособлен к обработке изображений, хранению видеоматериалов, видеоигр и т.п. В связи с этим компьютер, на котором установлен видеосервер, должен иметь высокую производительность и большую память.

* **Поисковый** *сервер* предназначен для поиска информации в Internet.
* **Почтовый** *сервер* предоставляет услуги в ответ на запросы, присланные по электронной почте.
* **Сервер WWW** предназначен для работы в Internet.
* **Сервер баз данных** выполняет обработку запросов к базам данных.
* **Сервер защиты данных** предназначен для обеспечения безопасности данных (содержит, например, средства для идентификации паролей).
* **Сервер приложений** предназначен для выполнения прикладных процессов. С одной стороны взаимодействует с *клиентами*, получая задания, а с другой – работает с базами данных, подбирая необходимые для обработки данные.
* **Сервер удаленного доступа** обеспечивает коллективный удаленный доступ к данным.
* **Файловый сервер** обеспечивает функционирование распределенных ресурсов, предоставляет услуги поиска, хранения, *архивирования*данных и возможность одновременного доступа к ним нескольких пользователей.

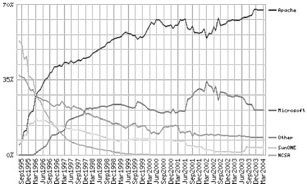
Обычно на компьютере- *сервере* работает сразу несколько программ- *серверов*. Одна занимается электронной почтой, другая распределением файлов, третья предоставляет web-страницы.

Из всех типов *серверов* нас в основном интересует *сервер WWW*. Часто его называют *web-сервером*, *http-сервером* или даже просто *сервером*. Что представляет собой *web-сервер*? Во-первых, это хранилище информационных ресурсов. Во-вторых, эти ресурсы хранятся и предоставляются пользователям в соответствии со стандартами *Internet* (такими, как *протокол передачи данных* *HTTP* ). Как предоставляются данные в соответствии с этим протоколом, мы рассмотрим чуть позже. Работа с документами *web-сервера* осуществляется при помощи браузера (например, IE, Opera или Mozilla Firefox), который отсылает *серверу* запросы, созданные в соответствии с протоколом *HTTP*. В процессе выполнения задания *сервер* может связываться с другими *серверами*.

Далее в ходе лекции, говоря " *сервер* ", мы будем подразумевать *web-сервер*.

В качестве примеров *web-серверов* можно привести *сервер* *Apache* группы *Apache*, *Internet* *Information* *Server* (*IIS*) компании Microsoft, SunOne фирмы *Sun* Microsystems, WebLogic фирмы BEA Systems, IAS (Inprise *Application Server*) фирмы Borland, WebSphere фирмы *IBM*, OAS (*Oracle* *Application Server*).

На [рис. 4.1](http://www.intuit.ru/studies/courses/42/42/lecture/27181?page=1#image.4.1) и в [таблице 4.1](http://www.intuit.ru/studies/courses/42/42/lecture/27181?page=1#table.4.1) приведена *статистика* использования различных *серверов* среди всех доменов *Internet* от NetCraft<http://news.netcraft.com/>.



**Рис. 4.1.** Статистика использования ведущих web-серверов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 4.1. Ведущие разработчики web-серверов | | | | | |
| **Разработчик** | **Февраль 2004** | **Проценты** | **Март 2004** | **Проценты** | **Изменение** |
| *Apache* | 31703884 | 67.21 | 32280582 | 67.20 | -0.01 |
| Microsoft | 9849971 | 20.88 | 10099760 | 21.02 | 0.14 |
| SunONE | 1657295 | 3.51 | 1651575 | 3.44 | -0.07 |
| Zeus | 755227 | 1.60 | 762716 | 1.59 | -0.01 |

Как видно из приведенной таблицы, *сервер* *Apache* занимает лидирующие позиции. Все, что мы когда-либо будем говорить о *web-серверах*, ориентировано на *Apache*, если не указано иное. О том, как установить его на свой *компьютер*, мы уже рассказывали в самой первой лекции. А теперь, как было обещано, обратимся к протоколу *HTTP*.

Далее мы рассмотрим наиболее распространенные *методы отправки запросов*.

#### Методы

Как уже говорилось, любой запрос *клиента* к *серверу* должен начинаться с указания метода. Метод сообщает о цели запроса *клиента*. Протокол *HTTP* поддерживает достаточно много методов, но реально используются только три: *POST* , *GET* и *HEAD* . Метод *GET*позволяет получить любые данные, идентифицированные с помощью URL в запросе ресурса. Если URL указывает на программу, то возвращается результат работы программы, а не ее текст (если, конечно, текст не есть результат ее работы). Дополнительная информация, необходимая для обработки запроса, встраивается в сам запрос (в адресную строку). При использовании метода *GET* в поле тела ресурса возвращается собственно затребованная информация (текст HTML-документа, например).

Существует разновидность метода *GET* – условный *GET* . Этот метод сообщает *серверу* о том, что на запрос нужно ответить, только если выполнено условие, содержащееся в поле if-Modified-Since заголовка запроса. Если говорить более точно, то тело ресурса передается в ответ на запрос, если этот ресурс изменялся после даты, указанной в if-Modified-Since.

Метод *HEAD* аналогичен методу *GET* , только не возвращает тело ресурса и не имеет условного аналога. Метод *HEAD* используют для получения информации о ресурсе. Это может пригодиться, например, при решении задачи тестирования гипертекстовых ссылок.

Метод *POST* разработан для передачи на *сервер* такой информации, как аннотации ресурсов, новостные и почтовые сообщения, данные для добавления в базу данных, т.е. для передачи информации большого объема и достаточно важной. В отличие от методов *GET* и *HEAD*, в *POST* передается тело ресурса, которое и является информацией, получаемой из полей форм или других источников ввода.

До сих пор мы только теоретизировали, знакомились с основными понятиями. Теперь пора научиться использовать все это на практике. Далее в лекции мы рассмотрим, как посылать запросы *серверу* и как обрабатывать его ответы.

### Использование HTML-форм для передачи данных на сервер

Как передавать данные *серверу*? Для этого в языке *HTML* есть специальная конструкция – формы. Формы предназначены для того, чтобы получать от пользователя информацию. Например, вам нужно знать логин и *пароль* пользователя для того, чтобы определить, на какие страницы сайта его можно допускать. Или вам необходимы личные данные пользователя, чтобы была возможность с ним связаться. Формы как раз и применяются для ввода такой информации. В них можно вводить текст или выбирать подходящие варианты из списка. Данные, записанные в форму, отправляются для обработки специальной программе (например, скрипту на *PHP*) на *сервере*. В зависимости от введенных пользователем данных эта *программа* может формировать различные web-страницы, отправлять запросы к базе данных, запускать различные приложения и т.п.

Разберемся с синтаксисом *HTML-форм*. Возможно, многие с ним знакомы, но мы все же повторим основные моменты, поскольку это важно.

Итак, для создания формы в языке *HTML* используется *тег* *FORM* . Внутри него находится одна или несколько команд *INPUT* . С помощью атрибутов *action* и *method* тега *FORM* задаются *имя программы*, которая будет обрабатывать данные формы, и метод запроса, соответственно. *Команда* *INPUT* определяет тип и различные характеристики запрашиваемой информации. *Отправка* данных формы происходит после нажатия кнопки *input* типа *submit* . Создадим форму для регистрации участников заочной школы программирования.

<h2>Форма для регистрации участников</h2>

<form action="1.php" method=POST> <!--создаем форму-->

<!--данные формы будет обрабатывать файл 1.php, при

отправке запроса будет использован метод POST-->

Имя <br><input type=text name="first\_name"

value="Введите Ваше имя"><br>

Фамилия <br><input type=text name="last\_name"><br>

E-mail <br><input type=text name="email"><br>

<p>

Выберите курс, который вы бы хотели посещать:<br>

<input type=radio name="kurs" value="PHP">PHP<br>

<input type=radio name="kurs" value="Lisp">Lisp<br>

<input type=radio name="kurs" value="Perl">Perl<br>

<input type=radio name="kurs" value="Unix">Unix<br></p>

<p>Что вы хотите, чтобы мы знали о вас? <BR>

<textarea name="comment" cols=32 rows=5></textarea></p>

<p><input name="confirm" type=checkbox

checked>Подтвердить получение <br>

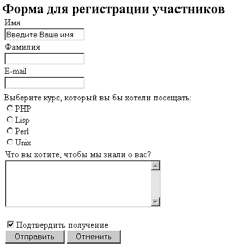
<input type=submit value="Отправить">

<input type=reset value="Отменить"></p>

</form>

Листинг 4.0. form.html

После обработки браузером этот *файл* будет выглядеть примерно так:



**Рис. 4.3.** Пример html-формы

Вот так создаются и выглядят *HTML-формы*. Будем считать, что мы научились или вспомнили, как их создавать. Как мы видим, в форме можно указывать *метод передачи данных*. Посмотрим, что будет происходить, если указать метод *GET* или *POST* , и в чем будет разница.

#### Для метода GET

При отправке данных формы с помощью метода *GET* содержимое формы добавляется к *URL* после знака вопроса в виде пар имя=значения, объединенных с помощью амперсанда &:

action?name1=value1&name2=value2&name3=value3

Здесь *action* – это *URL* -адрес программы, которая должна обрабатывать форму (это либо программа, заданная в атрибуте *action* тега *form* , либо сама текущая программа, если этот атрибут опущен). Имена name1, name2, name3 соответствуют именам элементов формы, а value1, value2, value3 – значениям этих элементов. Все специальные символы, включая = и &, в именах или значениях этих параметров будут закодированы. Поэтому не стоит использовать в названиях или значениях элементов формы эти символы и символы кириллицы в идентификаторах.

Если в поле для ввода ввести какой-нибудь служебный символ, то он будет передан в его шестнадцатеричном коде, например, символ $заменится на %24. Так же передаются и русские буквы.

Для полей ввода текста и пароля (это элементы *input* с атрибутом type=text и type=password ), значением будет то, что введет пользователь. Если пользователь ничего не вводит в такое поле, то в *строке запроса* будет присутствовать элемент name=, где nameсоответствует имени этого элемента формы.

Для кнопок типа checkbox и *radio button* значение value определяется атрибутом VALUE в том случае, когда кнопка отмечена. Не отмеченные кнопки при составлении *строки запроса* игнорируются целиком. Несколько кнопок типа checkbox могут иметь один атрибут NAME (и различные VALUE ), если это необходимо. Кнопки типа *radio button* предназначены для одного из всех предложенных вариантов и поэтому должны иметь одинаковый атрибут NAME и различные атрибуты VALUE.

В принципе создавать *HTML-форму* для передачи данных методом *GET* не обязательно. Можно просто добавить в строку *URL* нужные переменные и их значения.

http://phpbook.info/test.php?id=10&user=pit

Пример 4.1. Передача данных в URL

В связи с этим у передачи данных методом *GET* есть один существенный недостаток – любой может подделать значения параметров. Поэтому не советуем использовать этот метод для доступа к защищенным паролем страницам, для передачи информации, влияющей на безопасность работы программы или *сервера*. Кроме того, не стоит применять метод *GET* для передачи информации, которую не разрешено изменять пользователю.

Несмотря на все эти недостатки, использовать метод *GET* достаточно удобно при отладке скриптов (тогда можно видеть значения и имена передаваемых переменных) и для передачи параметров, не влияющих на безопасность.

#### Для метода POST

Содержимое формы кодируется точно так же, как для метода *GET* (см. выше), но вместо добавления строки к *URL* содержимое запроса посылается блоком данных как часть операции *POST* . Если присутствует атрибут ACTION, то значение *URL*, которое там находится, определяет, куда посылать этот блок данных. Этот метод, как уже отмечалось, рекомендуется для передачи больших по объему блоков данных.

Информация, введенная пользователем и отправленная *серверу* с помощью метода *POST* , подается на *стандартный ввод* программе, указанной в атрибуте *action* , или текущему скрипту, если этот атрибут опущен. Длина посылаемого файла передается в *переменной окружения* CONTENT\_LENGTH, а тип данных – в переменной CONTENT\_TYPE.

Передать данные методом *POST* можно только с помощью *HTML-формы*, поскольку данные передаются в теле запроса, а не в заголовке, как в *GET* . Соответственно и изменить значение параметров можно, только изменив значение, введенное в форму. При использовании *POST* пользователь не видит передаваемые *серверу* данные.

Основное преимущество *POST* запросов – это их большая безопасность и функциональность по сравнению с GET-запросами. Поэтому метод *POST* чаще используют для передачи важной информации, а также информации большого объема. Тем не менее не стоит целиком полагаться на безопасность этого механизма, поскольку данные *POST* запроса также можно подделать, например создав html-файл на своей машине и заполнив его нужными данными. Кроме того, не все *клиенты* могут применять метод *POST* , что ограничивает варианты его использования.

При отправке данных на *сервер* любым методом передаются не только сами данные, введенные пользователем, но и ряд переменных, называемых *переменными окружения*, характеризующих *клиента*, историю его работы, пути к файлам и т.п. Вот некоторые из *переменных окружения*:

* ***REMOTE\_ADDR*** – IP-адрес хоста (компьютера), отправляющего запрос;
* ***REMOTE\_HOST*** – *имя хоста*, с которого отправлен запрос;
* ***HTTP\_REFERER*** – адрес страницы, ссылающейся на текущий скрипт;
* ***REQUEST\_METHOD*** – метод, который был использован при отправке запроса;
* ***QUERY\_STRING*** – информация, находящаяся в *URL* после знака вопроса;
* ***SCRIPT\_NAME*** – виртуальный путь к программе, которая должна выполняться;
* ***HTTP\_USER\_AGENT*** – информация о браузере, который использует *клиент*

### Обработка запросов с помощью PHP

До сих пор мы упоминали только, что запросы *клиента* обрабатываются на *сервере* с помощью специальной программы. На самом деле эту программу мы можем написать сами, в том числе и на языке *PHP*, и она будет делать с полученными данными все, что мы захотим. Для того, чтобы написать эту программу, необходимо познакомиться с некоторыми правилами и инструментами, предлагаемыми для этих целей *PHP*.

Внутри *PHP-скрипта* имеется несколько способов получения доступа к данным, переданным *клиентом* по протоколу *HTTP*. До версии *PHP*4.1.0 *доступ* к таким данным осуществлялся по именам переданных переменных (напомним, что данные передаются в виде пар "*имя переменной*, символ "=", *значение* переменной"). Таким образом, если, например, было передано first\_name=Nina, то внутри скрипта появлялась *переменная* $first\_name со значением Nina. Если требовалось различать, каким методом были переданы данные, то использовались ассоциативные массивы *$HTTP\_POST\_VARS* и *$HTTP\_GET\_VARS* , ключами которых являлись имена переданных переменных, а значениями – соответственно значения этих переменных. Таким образом, если пара first\_name = Nina передана методом *GET* , то $*HTTP\_GET\_VARS*["first\_name"]="Nina".

Использовать в программе имена переданных переменных напрямую небезопасно. Поэтому было решено начиная с *PHP* 4.1.0 задействовать для обращения к переменным, переданным с помощью HTTP-запросов, специальный *массив* – *$\_REQUEST* . Этот *массив* содержит данные, переданные методами *POST* и *GET* , а также с помощью *HTTP* cookies. Это суперглобальный ассоциативный *массив*, т.е. его значения можно получить в любом месте программы, используя в качестве ключа имя соответствующей переменной (элемента формы).

**Пример 4.2**. Допустим, мы создали форму для регистрации участников заочной школы программирования, как в приведенном выше примере. Тогда в файле 1.php, обрабатывающем эту форму, можно написать следующее:

<?php

$str = "Здравствуйте,

".$\_REQUEST["first\_name"]. "

".$\_REQUEST["last\_name"]."! <br>";

$str .="Вы выбрали для изучения курс по

".$\_REQUEST["kurs"];

echo $str;

?>

Пример 4.2. Файл 1.php, обрабатывающий форму form.html

Тогда, если в форму мы ввели имя "Вася", фамилию "Петров" и выбрали среди всех курсов курс по *PHP*, на экране браузера получим такое сообщение:

Здравствуйте, Вася Петров!

Вы выбрали для изучения курс по PHP

После введения массива *$\_REQUEST* массивы *$HTTP\_POST\_VARS* и *$HTTP\_GET\_VARS* для однородности были переименованы в *$\_POST* и *$\_GET* соответственно, но сами они из обихода не исчезли из соображений совместимости с предыдущими версиями *PHP*. В отличие от своих предшественников, массивы *$\_POST* и *$\_GET* стали суперглобальными, т.е. доступными напрямую и внутри функций и методов.

Приведем пример использования этих массивов. Допустим, нам нужно обработать форму, содержащую элементы ввода с именами first\_name, last\_name, kurs (например, форму form.html, приведенную выше). Данные были переданы методом *POST* , и данные, переданные другими методами, мы обрабатывать не хотим. Это можно сделать следующим образом:

<?php

$str = "Здравствуйте,

".$\_POST ["first\_name"]."

".$\_POST ["last\_name"] ."! <br>";

$str .= "Вы выбрали для изучения курс по ".

$\_POST["kurs"];

echo $str;

?>

Тогда на экране браузера, если мы ввели имя "Вася", фамилию "Петров" и выбрали среди всех курсов курс по *PHP*, увидим сообщение, как в предыдущем примере:

Здравствуйте, Вася Петров!

Вы выбрали для изучения курс по PHP

Для того, чтобы сохранить возможность обработки скриптов более ранних версий, чем *PHP* 4.1.0, была введена *директиваregister\_globals* , разрешающая или запрещающая *доступ* к переменным непосредственно по их именам. Если в *файле настроек* *PHPпараметр* *register\_globals*=On, то к переменным, переданным *серверу* методами *GET* и *POST* , можно обращаться просто по их именам (т.е. можно писать $first\_name ). Если же *register\_globals*=Off, то нужно писать $*\_REQUEST*["first\_name"] или $*\_POST*["first\_name"], $*\_GET*["first\_name"], $*HTTP\_POST\_VARS*["first\_name"], $*HTTP\_GET\_VARS*["first\_name"]. С точки зрения безопасности эту директиву лучше отключать (т.е. *register\_globals*=Off ). При включенной директиве *register\_globals*перечисленные выше массивы также будут содержать данные, переданные *клиентом*.

Иногда возникает необходимость узнать *значение* какой-либо *переменной окружения*, например метод, использовавшийся при передаче запроса или IP-*адрес* компьютера, отправившего *запрос*. Получить такую информацию можно с помощью функции *getenv()* . Она возвращает *значение* *переменной окружения*, имя которой передано ей в качестве параметра.

<?

echo getenv('REQUEST\_METHOD');

// возвратит использованный метод

echo getenv ('REMOTE\_ADDR');

// выведет IP-адрес пользователя,

// пославшего запрос

?>

Пример 4.3. Использование функции getenv()

Как мы уже говорили, если используется метод *GET* , то данные передаются добавлением *строки запроса* в виде пар "имя\_переменной=*значение* к *URL* -адресу ресурса". Все, что записано в *URL* после знака вопроса, можно получить с помощью команды

getenv('QUERY\_STRING');

Благодаря этому можно по методу *GET* передавать данные в каком-нибудь другом виде. Например, указывать только значения нескольких параметров через знак плюс, а в скрипте разбирать строку запроса на части или можно передавать *значение* всего одного параметра. В этом случае в массиве *$\_GET* появится *пустой элемент* с ключом, равным этому значению (всей *строке запроса*), причем символ " + ", встретившийся в *строке запроса*, будет заменен на подчеркивание " \_ ".

Методом *POST* данные передаются только с помощью форм, и *пользователь* ( *клиент* ) не видит, какие именно данные отправляются *серверу*. Чтобы их увидеть, *хакер* должен подменить нашу форму своей. Тогда *сервер* отправит результаты обработки неправильной формы не туда, куда нужно. Чтобы этого избежать, можно проверять *адрес* страницы, с которой были посланы данные. Это можно сделать опять же с помощью функции *getenv()* :

getenv('HTTP\_REFERER');

Теперь самое время решить задачу, сформулированную в начале лекции.

#### Пример обработки запроса с помощью PHP

Напомним, в чем состояла задача, и уточним ее формулировку. Нужно написать форму для регистрации участников заочной школы программирования и после регистрации отправить участнику сообщение. Мы назвали это сообщение универсальным письмом, но оно будет немного отличаться от того письма, которое мы составили на предыдущей лекции. Здесь мы также не будем отправлять что-либо по электронной почте, дабы не уподобляться спамерам, а просто сгенерируем это сообщение и выведем его на экран браузера. Начальный вариант формы регистрации мы уже приводили выше. Изменим его таким образом, чтобы каждый регистрирующийся мог выбрать сколько угодно курсов для посещения, и не будем подтверждать получение регистрационной формы.

<h2>Форма для регистрации студентов</h2>

<form action="1.php" method=POST>

Имя <br><input type=text name="first\_name"

value="Введите Ваше имя"><br>

Фамилия <br><input type=text name="last\_name"><br>

E-mail <br><input type=text name="email"><br>

<p> Выберите курс, который вы бы хотели посещать:<br>

<input type=checkbox name='kurs[]' value='PHP'>PHP<br>

<input type=checkbox name='kurs[]' value='Lisp'>Lisp<br>

<input type=checkbox name='kurs[]' value='Perl'>Perl<br>

<input type=checkbox name='kurs[]' value='Unix'>Unix<br>

<P>Что вы хотите, чтобы мы знали о вас? <BR>

<textarea name="comment" cols=32 rows=5></textarea>

<input type=submit value="Отправить">

<input type=reset value="Отменить">

</form>

Листинг 4.4. form\_final.html

Здесь все достаточно просто и понятно. Единственное, что можно отметить, – это способ передачи значений элемента checkbox. Когда мы пишем в имени элемента kurs[], это значит, что первый отмеченный элемент checkbox будет записан в первый элемент массива kurs, второй отмеченный checkbox – во второй элемент массива и т.д. Можно, конечно, просто дать разные имена элементам checkbox, но это усложнит обработку данных, если курсов будет много.

Скрипт, который все это будет разбирать и обрабатывать, называется 1.php (форма ссылается именно на этот файл, что записано в ее атрибуте *action* ). По умолчанию используется для передачи метод *GET* , но мы указали *POST* . По полученным сведениям от зарегистрировавшегося человека, скрипт генерирует соответствующее сообщение. Если человек выбрал какие-то курсы, то ему выводится сообщение о времени их проведения и о лекторах, которые их читают. Если человек ничего не выбрал, то выводится сообщение о следующем собрании заочной школы программистов (ЗШП).

<?

// создадим массивы соответствий курс-время его

// проведения и курс-его лектор

$times = array("PHP"=>"14.30","Lisp"=>"12.00",

"Perl"=>"15.00","Unix"=>"14.00");

$lectors = array("PHP"=>"Василий Васильевич",

"Lisp"=>"Иван Иванович", "Perl"=>"Петр Петрович", "Unix"=>"Семен Семенович");

define("SIGN","С уважением, администрация");

// определяем подпись письма как константу

define("MEETING\_TIME","18.00");

// задаем время собрания студентов

$date = "12 мая"; // задаем дату проведения лекций

//начинаем составлять текст сообщения

$str = "Здравствуйте, уважаемый " . $\_POST["first\_name"]

. " " . $\_POST["last\_name"]."!<br>";

$str .= "<br>Сообщаем Вам, что ";

$kurses = $\_POST["kurs"]; // сохраним в этой переменной

// список выбранных курсов

if (!isset($kurses)) { // если не выбран ни один курс

$event = "следующее собрание студентов";

$str .= "$event состоится $date ". MEETING\_TIME . "<br>";

} else { // если хотя бы один курс выбран

$event = "выбранные Вами лекции состоятся $date <ul>";

//функция count вычисляет число элементов в массиве

$lect = "";

for ($i=0;$i<count($kurses);$i++){

// для каждого выбранного курса

$k = $kurses[$i]; // запоминаем название курса

$lect = $lect . "<li>лекция по $k в $times[$k]";

// составляем сообщение

$lect .= " (Ваш лектор, $lectors[$k])";

}

$event = $event . $lect . "</ul>";

$str .= "$event";

}

$str .= "<br>". SIGN; // добавляем подпись

echo $str; // выводим сообщение на экран

?>

Листинг 4.5. Скрипт 1.php, обрабатывающий форму form\_final.html

### Заключение

Подведем итоги. Мы научились отличать *клиента* от *сервера* и *компьютер*- *сервер* от программы- *сервера*, познакомились с основными методами, используемыми для передачи данных на *сервер*, изучили средства, предлагаемые языком *PHP* для обработки клиентских запросов, и испробовали их на практике. В принципе этого достаточно для того, чтобы создавать клиент-серверные приложения.

ФУНКЦИИ В PHP

В этой лекции будут рассматриваться вопросы создания и использования функций в *PHP*. Говоря "функции", мы не имеем в виду все существующие в *PHP* функции, а лишь функции, определяемые пользователем. Мы рассмотрим способы задания таких функций, методы передачи аргументов, использование аргументов со значением по умолчанию и значения, возвращаемые функцией.

В качестве примера создадим web-интерфейс для генерации *html-формы*. То есть *пользователь* выбирает, не прибегая к программированию, какие элементы формы нужно создать, и характеристики этих элементов, а наша *программа* генерирует нужную форму.

#### Функции, определяемые пользователем

Для чего нужны функции? Чтобы ответить на этот вопрос, нужно понять, что вообще представляют собой функции. В программировании, как и в математике, функция есть *отображение множества* ее аргументов на множество ее значений. То есть функция для каждого набора значений аргумента возвращает какие-то значения, являющиеся результатом ее работы. Зачем нужны функции, попытаемся объяснить на примере. Классический пример функции в программировании – это функция, вычисляющая значение факториала числа. То есть мы задаем ей число, а она возвращает нам его факториал. При этом не нужно для каждого числа, факториал которого мы хотим получить, повторять один и тот же код – достаточно просто вызвать функцию с аргументом, равным этому числу.

Функция вычисления факториала натурального числа

<?php

function fact($n){

if ($n==0) return 1;

else return $fact = $n \* fact($n-1);

}

echo fact(3);

// можно было бы написать echo (3\*2);

// но если число большое,

echo fact(50);

// то удобнее пользоваться функцией,

// чем писать echo (50\*49\*48\*...\*3\*2);

?>

Таким образом, когда мы осуществляем действия, в которых прослеживается зависимость от каких-то данных, и при этом, возможно, нам понадобится выполнять такие же действия, но с другими исходными данными, удобно использовать механизм функций – оформить блок действий в виде *тела функции*, а меняющиеся данные – в качестве ее параметров.

Посмотрим, как в общем виде выглядит задание (объявление) функции. Функция может быть определена с помощью следующего синтаксиса:

function Имя\_функции (параметр1, параметр2,

... параметрN){

Блок\_действий

return "значение, возвращаемое функцией";

}

Если прямо так написать в php-программе, то работать ничего не будет. Во-первых, Имя\_функции и имена *параметров функции* ( параметр1, параметр2 и т.д.) должны соответствовать правилам наименования в PHP (и русских символов в них лучше не использовать). Имена функций нечувствительны к регистру. Во-вторых, *параметры функции* – это переменные языка, поэтому перед названием каждого из них должен стоять знак $. Никаких многоточий ставить в списке параметров нельзя. В-третьих, вместо слов блок\_действий в *теле функции* должен находиться любой правильный PHP-код (не обязательно зависящий от параметров). И наконец, после ключевого слова *return* должно идти корректное php-выражение (что-либо, что имеет значение). Кроме того, у функции может и не быть параметров, как и возвращаемого значения. Пример правильного *объявления функции* – функция вычисления факториала, приведенная выше.

Как происходит вызов функции? Указывается имя функции и в круглых скобках список значений ее параметров, если таковые имеются:

<?php

Имя\_функции ("значение\_для\_параметра1",

"значение\_для\_параметра2",...);

// пример вызова функции – вызов функции

// вычисления факториала приведен выше,

// там для вычисления факториала числа 3

// мы писали: fact(3);

// где fact – имя вызываемой функции,

// а 3 – значение ее параметра с именем $n

?>

Когда можно вызывать функцию? Казалось бы, странный вопрос. Функцию можно вызвать после ее определения, т.е. в любой строке программы ниже блока function f\_name(){...}. В *PHP3* это было действительно так. Но уже в *PHP4* такого требования нет. Все дело в том, как интерпретатор обрабатывает получаемый код. Единственное исключение составляют функции, определяемые условно (внутри условных операторов или других функций). Когда функция определяется таким образом, ее определение должно предшествовать ее вызову.

<?php

$make = true;

/\* здесь нельзя вызвать Make\_event();

потому что она еще не существует, но можно

вызвать Save\_info() \*/

Save\_info("Вася","Иванов",

"Я выбрал курс по PHP");

if ($make){

// определение функции Make\_event()

function Make\_event(){

echo "<p>Хочу изучать Python<br>";

}

}

// теперь можно вызывать Make\_event()

Make\_event();

// определение функции Save\_info

function Save\_info($first, $last, $message){

echo "<br>$message<br>";

echo "Имя: ". $first . " ". $last . "<br>";

}

Save\_info("Федя","Федоров",

"А я выбрал Lisp");

// Save\_info можно вызывать и здесь

?>

Пример 5.1. Определение функции внутри условного оператора

Если функция однажды определена в программе, то переопределить или удалить ее позже нельзя. Несмотря на то, что имена функций нечувствительны к регистру, лучше вызывать функцию по тому же имени, каким она была задана в определении.

<?php

/\* нельзя сохранить данные, т.е. вызвать

функцию DataSave() до того, как выполнена

проверка их правильности, т.е. вызвана

функция DataCheck() \*/

DataCheck();

DataSave();

function DataCheck(){

// проверка правильности данных

function DataSave(){

// сохраняем данные

}

}

?>

Пример 5.2. Определение функции внутри функции

Рассмотрим подробнее аргументы функций, их назначение и использование.

#### Аргументы функций

У каждой функции может быть, как мы уже говорили, *список аргументов*. С помощью этих аргументов в функцию передается различная информация (например, значение числа, факториал которого надо подсчитать). Каждый аргумент представляет собой переменную или константу.

С помощью аргументов данные в функцию можно передавать тремя различными способами. Это передача аргументов по значению (используется по умолчанию), по ссылке и *задание значения аргументов по умолчанию*. Рассмотрим эти способы подробнее.

Когда аргумент передается в функцию по значению, изменение значения аргумента внутри функции не влияет на его значение вне функции. Чтобы позволить функции изменять ее аргументы, их нужно передавать по ссылке. Для этого в определении функции перед именем аргумента следует написать знак амперсанд "&".

<?php

// напишем функцию, которая бы добавляла

// к строке слово checked

function add\_label(&$data\_str){

$data\_str .= "checked";

}

$str = "<input type=radio name=article ";

// пусть имеется такая строка

echo $str ."<br>";

// выведет элемент формы –

// не отмеченную радио кнопку

add\_label($str);

// вызовем функцию

echo $str ."><br>";

// это выведет уже отмеченную

// радио кнопку

?>

Пример 5.3. Передача аргументов по ссылке

В функции можно определять значения аргументов, используемые по умолчанию. Само значение по умолчанию должно быть константным выражением, а не переменной и не представителем класса или вызовом другой функции.

У нас есть функция, создающая *информационное сообщение*, подпись к которому меняется в зависимости от значения переданного ей параметра. Если значение параметра не задано, то используется подпись "Оргкомитет.".

<?php

function Message($sign="Оргкомитет."){

// здесь параметр sign имеет по умолчанию значение "Оргкомитет"

echo "Следующее собрание состоится завтра.<br>";

echo $sign . "<br>";

}

Message();

// вызываем функцию без параметра.

// В этом случае подпись – это Оргкомитет

Message("С уважением, Вася.");

// В этом случае подпись

// будет "С уважением, Вася."

?>

Пример 5.4. Значения аргументов по умолчанию

Результатом работы этого скрипта будет:

Следующее собрание состоится завтра.

Оргкомитет.

Следующее собрание состоится завтра.

С уважением, Вася.

Если у функции несколько параметров, то те аргументы, для которых задаются значения по умолчанию, должны быть записаны после всех остальных аргументов в определении функции. В противном случае появится ошибка, если эти аргументы будут опущены при вызове функции.

Например, мы хотим внести описание статьи в каталог. Пользователь должен ввести такие характеристики статьи, как ее название, автора и краткое описание. Если пользователь не вводит имя автора статьи, считаем, что это Иванов Иван.

<?php

function Add\_article($title, $description,

$author="Иванов Иван"){

echo "Заносим в каталог статью: $title,";

echo "автор $author";

echo "<br>Краткое описание: ";

echo "$description <br>";

}

Add\_article("Информатика и мы",

"Это статья про информатику ...",

"Петров Петр");

Add\_article("Кто такие хакеры",

"Это статья про хакеров ...");

?>

В результате работы скрипта получим следующее

Заносим в каталог статью: Информатика и мы,

автор Петров Петр.

Краткое описание:

Это статья про информатику...

Заносим в каталог статью: Кто такие хакеры,

автор Иванов Иван.

Краткое описание:

Это статья про хакеров...

Если же мы напишем вот так:

<?php

function Add\_article($author="Иванов Иван",

$title, $description){

// ...действия как в предыдущем примере

}

Add\_article("Кто такие хакеры",

"Это статья про хакеров...");

?>

То в результате получим:

Warning: Missing argument 3 for

add\_article() in

c:\users\nina\tasks\func\def\_bad.php

on line 2

#### Использование переменных внутри функции

##### Глобальные переменные

Чтобы использовать внутри функции переменные, заданные вне ее, эти переменные нужно объявить как глобальные. Для этого в *теле функции* следует перечислить их имена после ключевого слова *global*:

global $var1, $var2;

<?

$a=1;

function Test\_g(){

global $a;

$a = $a\*2;

echo 'в результате работы функции $a=',$a;

}

echo 'вне функции $a=',$a,', ';

Test\_g();

echo "<br>";

echo 'вне функции $a=',$a,', ';

Test\_g();

?>

Пример 5.7. Глобальные переменные

В результате работы этого скрипта получим:

вне функции $a=1, в результате работы

функции $a=2

вне функции $a=2, в результате работы

функции $a=4

Когда переменная объявляется как глобальная, фактически создается ссылка на *глобальную переменную*. Поэтому такая запись эквивалентна следующей (массив $GLOBALS содержит все переменные, глобальные относительно текущей области видимости):

$var1 = &$GLOBALS["var1"];

$var2 = &$GLOBALS["var2"];

Это значит, например, что удаление переменной $var1 не удаляет *глобальной переменной* $GLOBALS["var1"].

##### Статические переменные

Чтобы использовать переменные только внутри функции, при этом сохраняя их значения и после выхода из функции, нужно объявить эти переменные как статические. ***Статические переменные*** видны только внутри функции и не теряют своего значения, если выполнение программы выходит за *пределы функции*. Объявление таких переменных производится с помощью ключевого слова *static*:

static $var1, $var2;

*Статической переменной* может быть присвоено любое значение, но не ссылка.

<?

function Test\_s(){

static $a = 1;

// нельзя присваивать выражение или ссылку

$a = $a\*2;

echo $a;

}

Test\_s(); // выведет 2

echo $a; // ничего не выведет, так как

// $a доступна только

// внутри функции

Test\_s(); // внутри функции $a=2, поэтому

// результатом работы функции

// будет число 4

?>

Пример 5.8. Использование статической переменной

#### Возвращаемые значения

Все функции, приведенные выше в качестве примеров, выполняли какие-либо действия. Кроме подобных действий, любая функция может возвращать как результат своей работы какое-нибудь значение. Это делается с помощью утверждения *return*. Возвращаемое значение может быть любого типа, включая списки и объекты. Когда интерпретатор встречает команду *return* в *теле функции*, он немедленно прекращает ее исполнение и переходит на ту строку, из которой была вызвана функция.

Например, составим функцию, которая возвращает возраст человека. Если человек не умер, то возраст считается относительно текущего года.

<?php

/\* если второй параметр вычисляется

как true, то он рассматривается как

дата смерти, \*/

function Age($birth, $is\_dead){

if ($is\_dead) return $is\_dead-$birth;

else return date("Y")-$birth;

}

echo Age(1971, false); // для 2009 года выведет 38

echo Age(1971, 2001); // выведет 30

?>

В этом примере можно было и не использовать функцию *return*, а просто заменить ее функцией вывода echo. Однако если мы все же делаем так, что функция возвращает какое-то значение (в данном случае возраст человека), то в программе мы можем присвоить любой переменной значение этой функции:

$an\_age = Age(1981, 2004);

В результате работы функции может быть возвращено только одно значение. Несколько значений можно получить, если возвращать список значений (*одномерный массив*). Допустим, мы хотим получить полный возраст человека с точностью до дня.

<?php

function Full\_age($b\_day, $b\_month, $b\_year)

{

$y = date("Y");

$m = intval(date("m"));

$d = intval(date("d"));

$b\_month = intval($b\_month);

$b\_day = intval($b\_day);

$b\_year = intval($b\_year);

$day = ($b\_day > $d ? 30 - $b\_day + $d : $d - $b\_day);

$tmpMonth = ($b\_day > $d ? -1 : 0);

$month = ($b\_month > $m + $tmpMonth ? 12 - $b\_month +

$tmpMonth + $m : $m+$tmpMonth - $b\_month);

$tmpYear = ($b\_month > $m + $tmpMonth ? -1 : 0);

if ($b\_year > $y + $tmpYear)

{

$year = 0; $month = 0; $day = 0;

}

else

{

$year = $y + $tmpYear - $b\_year;

}

return array ($day,$month,$year);

}

$age = Full\_age("29","06","1986");

echo "Вам $age[2] лет, $age[1] месяцев и $age[0] дней";

?>

Когда функция возвращает несколько значений для их обработки в программе, удобно использовать языковую конструкцию *list* (), которая позволяет одним действием присвоить значения сразу нескольким переменным. Например, в предыдущем примере, оставив без изменения функцию, обработать возвращаемые ей значения можно было так:

<?

// задание функции Full\_age()

list($day,$month,$year) = Full\_age("07",

"08","1974");

echo "Вам $year лет, $month месяцев и

$day дней";

?>

Вообще конструкцию *list* () можно использовать для присвоения переменным значений элементов любого массива.

<?

$arr = array("first","second");

list($a,$b) = $arr;

// переменной $a присваивается первое

// значение массива, $b – второе

echo $a," ",$b;

// выведет строку "first second"

?>

Пример 5.9. Использование list()

##### Возвращение ссылки

В результате своей работы функция также может возвращать ссылку на какую-либо переменную. Это может пригодиться, если требуется использовать функцию для того, чтобы определить, какой переменной должна быть присвоена ссылка. Чтобы получить из функции ссылку, нужно при объявлении перед ее именем написать знак амперсанд ( & ) и каждый раз при вызове функции перед ее именем тоже писать амперсанд ( & ). Обычно функция возвращает ссылку на какую-либо *глобальную переменную* (или ее часть – ссылку на элемент глобального массива), ссылку на *статическую переменную* (или ее часть) или ссылку на один из аргументов, если он был также передан по ссылке.

<?

$a = 3; $b = 2;

function & ref($par){

global $a, $b;

if ($par % 2 == 0) return $b;

else return $a;

}

$var =& ref(4);

echo $var, " и ", $b,"<br>";

//выведет 2 и 2

$b = 10;

echo $var, " и ", $b,"<br>";

// выведет 10 и 10

?>

Пример 5.10. Возвращение ссылки

При использовании синтаксиса ссылок в переменную $var нашего примера не копируется значение переменной $b возвращенной функцией $ref, а создается ссылка на эту переменную. То есть теперь переменные $var и $b идентичны и будут изменяться одновременно.

#### Переменные функции

PHP поддерживает концепцию *переменных функций*. Это значит, что если имя переменной заканчивается круглыми скобками, то PHP ищет функцию с таким же именем значения и пытается ее выполнить.

<?

/\* создадим две простые функции:

Add\_sign – добавляет подпись к строке и

Show\_text – выводит строку текста \*/

function Add\_sign($string,

$sign="С уважением, Петр"){

echo $string ." ".$sign;

}

function Show\_text(){

echo "Отправить сообщение по почте<br>";

}

$func = "Show\_text";

// создаем переменную со значением,

// равным имени функции Show\_text

$func();

// это вызовет функцию Show\_text

$func = "Add\_sign";

// создаем переменную со значением,

// равным имени функции Add\_sign

$func("Привет всем <br>");

// это вызовет функцию

// Add\_sign с параметром "Привет всем"

?>

Пример 5.11. Использование переменных функций

В этом примере функция Show\_text просто выводит строку текста. Казалось бы, зачем для этого создавать отдельную функцию, если существует специальная функция echo(). Дело в том, что такие функции, как echo(), print(), *unset*(), include() и т.п. нельзя использовать в качестве *переменных функций*. То есть если мы напишем:

<?

$func = "echo ";

$func("TEXT");

?>

то интерпретатор выведет ошибку:

Fatal error: Call to undefined function:

echo() in

c:\users\nina\tasks\func\var\_f.php on line 2

Поэтому для того, чтобы использовать любую из перечисленных выше функций как *переменную функцию*, нужно создать собственную функцию, что мы и сделали в предыдущем примере.

#### Внутренние (встроенные) функции

Говоря о функциях, определяемых пользователем, все же нельзя не сказать пару слов о *встроенных функциях*. С некоторыми из *встроенных функций*, такими как echo(), print(), date(), include(), мы уже познакомились. На самом деле все перечисленные функции, кроме date(), являются языковыми конструкциями. Они входят в ядро PHP и не требуют никаких дополнительных настроек и модулей. Функция date() тоже входит в состав ядра PHP и не требует настроек. Но есть и функции, для работы с которыми нужно установить различные библиотеки и подключить соответствующий модуль. Например, для использования функций работы с базой данных *MySql* следует скомпилировать PHP с поддержкой этого расширения. В последнее время наиболее распространенные расширения и соответственно их функции изначально включают в состав PHP так, чтобы с ними можно работать без каких бы то ни было дополнительных настроек интерпретатора.

#### Решение задачи

Напомним, в чем состоит задача. Мы хотим написать интерфейс, который позволял бы создавать *html-формы*. Пользователь выбирает, какие элементы и в каком количестве нужно создать, придумывает им названия, а наша программа сама генерирует требуемую форму.

Разобьем задачу на несколько подзадач: выбор типов элементов ввода и их количества, создание названий элементов ввода и обработка полученных данных, т.е. непосредственно генерация формы. Первая задача достаточно проста: нужно написать соответствующую форму, например подобную приведенной ниже ():

<form action="ask\_names.php">

Создать элемент "строка ввода текста": <input

type=checkbox name=types[]

value=string><br>

Количество элементов: <input type=text

name=numbers[string]

size=3><br>

<br>

Создать элемент "текстовая область": <input

type=checkbox

name=types[] value=text><br>

Количество элементов: <input type=text

name=numbers[text]

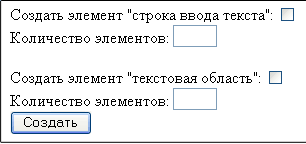
size=3><br>

<input type=submit value="Создать">

</form>

Листинг 5.12. task\_form.html

Когда мы пишем в имени элемента формы, например types[], это значит, что его имя – следующий элемент массива types. То есть у нас первый элемент формы ( "строка ввода текста" ) будет иметь имя types[0], а второй (текстовая область) – types[1]. В браузере task\_form.html будет выглядеть примерно так:



**Рис. 5.1.** Форма для выбора создаваемых элементов и их количества

После отправки данных этой формы мы получим информацию о том, какие элементы и сколько элементов каждого типа нужно создать. Следующий скрипт запрашивает названия для этих элементов:

<?

$file = "task.php";

/\* файл, который будет обрабатывать

сгенерированную этим скриптом форму \*/

function Ask\_names(){

// функция генерирует форму для

// ввода названий элементов ввода

global $file;

//объявляем, что хотим использовать эту

// переменную, заданную вне функции

if (isset($\_GET["types"])){

$st = '<form action="'.$file.'">';

foreach ($\_GET["types"] as $k => $type){

/\* перебираем все типы элементов,

которые нужно создать \*/

$num = $\_GET["numbers"][$type];

// сколько элементов каждого типа нужно

for ($i=1;$i<=$num;$i++){

// создаем $num строк для ввода

$st.= "Введите имя $i-го элемента типа $type: ";

$st.= "<input type=text name=names[$type][]><br>";

}

// сохраняем тип и число необходимых

// элементов ввода этого типа

$st.= "<input type=hidden name=types[] value=$type>";

$st.= "<input type=hidden name=numbers[] value=$num><br>";

}

$st .= "<input type=submit name=send value=send></form>";

return $st;

// в переменной $st содержится код формы

// для запроса имен

} else echo "Select type";

}

echo Ask\_names();

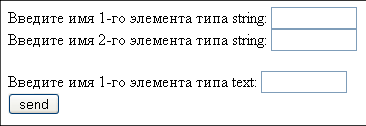
// вызываем функцию и выводим

// результаты ее работы

?>

Листинг 5.13. ask\_names.php

Допустим, нужно создать два элемента типа "текстовая строка" и один элемент типа "текстовая область", как и отмечено в форме выше. Тогда скрипт ask\_names.php обработает ее таким образом, что мы получим такую форму:



**Рис. 5.2.** Форма для ввода названий создаваемых элементов

Введем в эту форму, например, строки "Название", "Автор" и "Краткое содержание". Эти данные будет обрабатывать скрипт task.php.

<?

$show\_file = "task\_show.php";

/\* файл, который будет обрабатывать данные

созданной этим файлом формы \*/

function Create\_element($type,$name){

// функция создает элемент ввода

// по типу и названию

$str="";

switch($type){

case "string":

$str .= "$name: <input type=text name=string[]><br>";

break;

case "text":

$str .= "$name: <textarea name=text[]></textarea><br>";

break;

}

return $str;

}

function Create\_form(){

// функция создает форму

// с нужными элементами

global $show\_file;

$str = '<form action="'.$show\_file.'">';

foreach ($\_GET["types"] as $k => $type){

// перебираем типы элементов

$num = $\_GET["numbers"][$k];

// число элементов этого типа

for ($i=1;$i<=$num;$i++){

$arr = $\_GET["names"][$type][$i-1];

// имя создаваемого элемента

$str .= Create\_element($type,$arr);

// вызываем функцию для

// создания элемента

}

}

$str .= "<input type=submit value=send></form>";

echo $str;

}

$crt = "Create\_form";

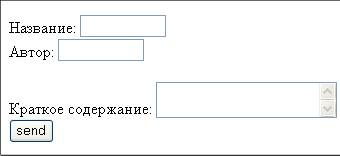
$crt(); // вызываем функцию создания

// формы Create\_form

?>

Листинг 5.14. task.php

Результатом работы этого скрипта с входными данными, приведенными выше, будет следующая форма:



**Рис. 5.3.** Пример формы, сгенерированной нашей программой

Подведем итоги. В этой лекции мы изучили функции, определяемые пользователем, их синтаксис и семантику, способы передачи их аргументов и возвращаемых значений. Кроме того, обсуждались способы задания и работы с функциями, имеющими переменное число аргументов и альтернативный способ вызова функции (с помощью переменной, значение которой есть имя функции). В следующей лекции будет рассмотрена объектная модель языка PHP.

**4.3 Работа с массивами и строками**

#### Тип array (массив)

***Массив*** в PHP представляет собой упорядоченную карту – *тип*, который преобразует *значения* в *ключи*. Этот *тип* оптимизирован в нескольких направлениях, поэтому вы можете использовать его как собственно *массив*, список (вектор), хеш-таблицу (являющуюся реализацией карты), стек, очередь и т.д. Поскольку вы можете иметь в качестве *значения* другой *массив* PHP, можно также легко эмулировать деревья.

Определить *массив* можно с помощью конструкции *array* () или непосредственно задавая *значения* его элементам.

##### Определение при помощи array()

array (key => value,

key1 => value1, ... )

Языковая конструкция ***array*** () принимает в качестве параметров пары *ключ* => ***значение***, разделенные запятыми. Символ =>устанавливает соответствие между *значением* и его *ключом*. *Ключ* может быть как *целым числом*, так и *строкой*, а *значение* может быть любого имеющегося в PHP *типа*. Числовой *ключ* *массива* часто называют индексом. Индексирование *массива* в PHP начинается с нуля. *Значение* элемента *массива* можно получить, указав после имени *массива* в *квадратных скобках* *ключ* искомого элемента. Если ***ключ*** *массива* представляет собой стандартную запись *целого числа*, то он рассматривается как число, в противном случае – как *строка*. Поэтому запись $a["1"] равносильна записи $a[1], так же как и $a["-1"] равносильно $a[-1].

<?php

$books = array ("php" =>

"PHP users guide",

12 => true);

echo $books["php"];

//выведет "PHP users guide"

echo $books[12];

//выведет 1

?>

Пример 2.9. Массивы в PHP

Если для элемента *ключ* не задан, то в качестве *ключа* берется максимальный числовой *ключ*, увеличенный на единицу. Если указать *ключ*, которому уже было присвоено какое-то *значение*, то оно будет перезаписано. Начиная с PHP 4.3.0, если максимальный *ключ* – отрицательное число, то следующим *ключом* *массива* будет ноль ( 0 ).

<?php

// массивы $arr и $arr1 эквивалентны

$arr = array(5 => 43, 32, 56, "b" => 12);

$arr1 = array(5 => 43, 6 => 32,

7 => 56, "b" => 12);

?>

Пример 2.10. Массивы в PHP

Если использовать в качестве *ключа* TRUE или FALSE, то его *значение* переводится соответственно в единицу и ноль *типа integer*. Если использовать *NULL*, то вместо *ключа* получим пустую *строку*. Можно использовать и саму пустую *строку* в качестве *ключа*, при этом ее надо брать в кавычки. Так что это не то же самое, что использование пустых *квадратных скобок*. Нельзя использовать в качестве *ключа* *массивы*и *объекты*.

##### Определение с помощью синтаксиса квадратных скобок

Создать *массив* можно, просто записывая в него *значения*. Как мы уже говорили, *значение* элемента *массива* можно получить с помощью *квадратных скобок*, внутри которых нужно указать его *ключ*, например, $book["php"]. Если указать новый *ключ* и новое *значение*, например, $book["new\_key"]="new\_value", то в *массив* добавится новый элемент. Если мы не укажем *ключ*, а только присвоим *значение*$book[]="new\_value", то новый элемент *массива* будет иметь числовой *ключ*, на единицу больший максимального существующего. Если *массив*, в который мы добавляем *значения*, еще не существует, то он будет *создан*.

<?

$books["key"]= value; // добавили в массив

// $books значение

// value с ключом key

$books[] = value1; /\* добавили в массив

значение value1 с

ключом 13, поскольку

максимальный ключ у

нас был 12 \*/

?>

Для того чтобы изменить конкретный элемент *массива*, нужно просто присвоить ему с его *ключом* новое *значение*. Изменить *ключ* элемента нельзя, можно только *удалить элемент* (пару *ключ* / *значение* ) и добавить новую. Чтобы ***удалить элемент*** *массива*, нужно использовать функцию ***unset*** () .

<?php

$books = array ("php" =>

"PHP users guide",

12 => true);

$books[] =

"Book about Perl"; // добавили элемент

// с ключом (индексом)

// 13 это эквивалентно

// $books[13] =

// "Book about Perl";

$books["lisp"] =

123456; /\* Это добавляет к массиву новый

элемент с ключом "lisp" и

значением 123456 \*/

unset($books[12]); // Это удаляет элемент

// c ключом 12 из массива

unset ($books); // удаляет массив полностью

?>

Заметим, что, когда используются пустые *квадратные скобки*, максимальный числовой *ключ* ищется среди *ключей*, существующих в *массиве*с момента последнего *переиндексирования*. ***Переиндексировать массив*** можно с помощью функции ***array\_values*** ().

<?php

$arr =

array ("a","b","c"); /\* Создаем массив

со значениями

"a", "b" и "c".

Поскольку ключи

не указаны, они

будут 0,1,2

соответственно \*/

print\_r($arr); // выводим массив (и ключи,

// и значения)

unset($arr[0]);

unset($arr[1]);

unset($arr[2]);

// удаляем из него все значения

print\_r($arr); // выводим массив (и ключи,

// и значения)

$arr[] = "aa"; // добавляем новый элемент

// в массив.

// Его индексом (ключом)

// будет 3, а не 0

print\_r($arr);

$arr =

array\_values($arr); // переиндексируем

// массив

$arr[] = "bb"; // ключом этого элемента

// будет 1

print\_r($arr);

?>

Пример 2.11. Переиндексация массива

Результатом работы этого скрипта будет:

Array ( [0] => a [1] => b [2] => c )

Array ( )

Array ( [3] => aa )

Array ( [0] => aa [1] => bb )

#### Операции с массивами

*Массив* – это тип данных, с данными этого типа должны быть определены *операции*. Какие же *операции* можно производить с *массивами*? *Массивы* можно *складывать* и *сравнивать*.

*Складывают массивы* с помощью стандартного оператора " + ". Вообще говоря, эту *операцию* по отношению к *массивам* точнее назвать объединением. Если у нас есть два *массива*, $a и $b, то результатом их *сложения* (объединения) будет *массив* $c, состоящий из элементов $a, к которым справа дописаны элементы *массива* $b. Причем, если встречаются совпадающие *ключи*, то в результирующий *массив* включается элемент из первого *массива*, т.е. из $a. Таким образом, если *складываются массивы* в языке PHP, от перемены мест слагаемых сумма меняется.

<?

$a = array("и"=>"Информатика",

"м"=>"Математика");

$b = array("и"=>"История","м"=>"Биология",

"ф"=>"Физика");

$c = $a + $b;

$d = $b + $a;

print\_r($c);

/\* получим: Array([и]=>Информатика

[м]=>Математика [ф]=>Физика) \*/

print\_r($d);

/\* получим: Array([и]=>История

[м]=>Биология [ф]=>Физика) \*/

?>

Пример 7.1. Сложение массивов

*Сравнивать массивы* можно, проверяя их равенство или неравенство либо эквивалентность или неэквивалентность. Равенство *массивов* – это когда совпадают все пары *ключ* / *значение* элементов *массивов*. Эквивалентность – когда кроме равенства *значений* и *ключей* элементов требуется еще, чтобы элементы в обоих *массивах* были записаны в одном и том же порядке. Равенство *значений* в PHP обозначается символом " == ", а эквивалентность – символом " === ".

<?

$a = array("и"=>"Информатика",

"м"=>"Математика");

$b = array("м"=>"Математика",

"и"=>"Информатика");

if ($a == $b) echo "Массивы равны и";

else echo "Массивы НЕ равны и ";

if ($a === $b) echo " эквивалентны";

else echo " НЕ эквивалентны";

// получим echo "Массивы равны и

НЕ эквивалентны"

?>

Пример 7.2. Сравнение массивов

Далее рассмотрим еще одну важную *операцию с массивом* – подсчет *количества его элементов*. Для ее реализации в PHP есть специальная функция.

##### Функция count

Не раз уже мы использовали функцию *count()* , чтобы вычислить *количество элементов массива*. На самом деле эта функция вычисляет число элементов в переменной вообще. Если применить ее к любой другой переменной, она возвратит 1. Исключение составляет переменная типа NULL – *count(NULL)* есть 0. Кроме того, применяя эту функцию к многомерному *массиву*, чтобы получить *число его элементов*, нужно использовать дополнительный параметр COUNT\_RECURSIVE.

<?

$del\_items = array("langs" => array(

"10"=>"Python", "12"=>"Lisp"),

"other"=>"Информатика");

echo count($del\_items) . "<br>";

// выведет 2

echo count($del\_items,COUNT\_RECURSIVE);

// выведет 4

?>

Пример 7.3. Применение функции count()

Мы не будем повторять все, что было сказано о *массивах* в предыдущих лекциях. В этой лекции мы рассмотрим некоторые встроенные функции для работы с *массивами*. И начнем мы с функций для *поиска значений в массиве*.

##### Функция in\_array

in\_array("искомое значение","массив",

["ограничение на тип"]);

позволяет установить, содержится ли в заданном *массиве* искомое *значение*. Если третий аргумент задан как true, то в *массиве* нужно найти элемент, совпадающий с искомым не только по *значению*, но и по типу. Если искомое *значение* – строка, то сравнение *чувствительно к регистру*.

Например, имеется *массив* неизученных нами языков программирования. Мы хотим узнать, содержится ли в этом *массиве* язык PHP. Напишем следующую программу:

<?php

$langs = array("Lisp","Python","Java",

"PHP","Perl");

if (in\_array("PHP",$langs,true))

echo "Надо бы изучить PHP<br>";

// выведет сообщение "Надо бы изучить PHP"

if (in\_array("php",$langs))

echo "Надо бы изучить php<br>";

// ничего не выведет, поскольку в массиве

// есть строка "PHP", а не "php"

?>

В качестве искомого *значения* этой функции может выступать и *массив*. Правда, это свойство было добавлено только начиная с PHP 4.2.0.

Например:

<?php

$langs = array("Lisp","Python",array("PHP","Java"),"Perl");

if (in\_array(array("PHP","Java"),$langs))

echo "Надо бы изучить PHP и Java<br>";

?>

##### Функция array\_search

Это еще одна функция для *поиска значения в массиве*. В отличие от *in\_array* в результате работы ***array\_search*** возвращает *значениеключа*, если элемент найден, и ложь – в противном случае. А вот синтаксис у этих функций одинаковый:

array\_search("искомое значение","массив",

["ограничение на тип"]);

Сравнение строк *чувствительно к регистру*, а если указан опциональный аргумент, то сравниваются еще и типы *значений*. До PHP 4.2.0, если искомое *значение* не было найдено, эта функция возвращала ошибку или пустое *значение* NULL.

**Пример 7.4**.Теперь, наоборот, пусть у нас есть *массив* языков программирования, которые мы знаем. Причем *ключом* каждого элемента является номер, указывающий, каким по счету был изучен этот язык.

<?php

$langs = array("","Lisp","Python","Java",

"PHP","Perl");

if (!array\_search("PHP",$langs))

echo "Надо бы изучить PHP<br>";

else {

$k = array\_search("PHP",$langs);

echo "PHP я изучила $k-м";

}

?>

Пример 7.4. Применение функции array\_search()

В результате мы получим строчку:

PHP я изучила 4-м

Очевидно, что эта функция более функциональна, чем *in\_array* , поскольку мы не только получаем информацию о том, что искомый элемент в *массиве* есть, но и узнаем, где именно в *массиве* он находится. А что будет, если искомых элементов в *массиве* несколько? В таком случае функция *array\_search()* вернет *ключ* первого из найденных элементов. Чтобы получить *ключи* всех элементов, нужно воспользоваться функцией ***array\_keys()*** .

##### Функция array\_keys

Функция *array\_keys()* выбирает все *ключи* *массива*. Но у нее имеется дополнительный аргумент, с помощью которого можно получить список *ключей* элементов с конкретным *значением*. Синтаксис этой функции таков:

array\_keys ("массив",

["значение для поиска"])

Функция *array\_keys()* возвращает как строковые, так и числовые *ключи* *массива*, организуя все *значения* в виде нового *массива* с числовыми индексами.

**Пример 7.5**.Мы записали *массив* языков, которые изучили. Список был длинным, и некоторые языки были записаны несколько раз. У нас возникло подозрение, что один из таких языков – Lisp. Давайте это проверим:

<?php

$langs =

array("Lisp","Python","Java","PHP",

"Perl","Lisp");

$lisp\_keys = array\_keys($langs,"Lisp");

echo "Lisp входит в массив ".

count($lisp\_keys) ." раза:<br>";

foreach ($lisp\_keys as $val){

echo "под номером $val <br>";

}

?>

Пример 7.5. Применение функции array\_keys()

В результате получим:

Lisp входит в массив 2 раза:

под номером 0

под номером 5

Функция *array\_keys()* , как и две предыдущие, зависит от регистра, т.е. элементов LISP в *массиве* она не обнаружит. *array\_keys()*появилась только в *PHP4*. В *PHP3* для реализации ее функциональности нужно придумывать свою функцию.

Если есть функция для получения всех *ключей* *массива*, то можно предположить, что существует и функция для получения всех *значениймассива*. Действительно, она существует. Это функция *array\_values*(массив). Все *значения* переданного ей *массива* записываются в новый *массив*, проиндексированный целыми числами, т.е. все *ключи* *массива* теряются, остаются только *значения*. Но вернемся к нашему примеру.

Итак, мы выяснили, что язык Lisp случайно упомянут в нашем *массиве* дважды. Поскольку изучить один язык дважды нельзя ("учил, но забыл" не считается), то нужно как-то избавиться от повторяющихся языков. Сделать это довольно просто с помощью функции *array\_unique()* .

##### Функция array\_unique

Функция *array\_unique(массив)* возвращает новый *массив*, в котором повторяющиеся элементы фигурируют в одном экземпляре. Таким образом, вместо нескольких одинаковых *значений* и их *ключей* мы имеем одно *значение*. Какой у него будет *ключ*? Как из нескольких *ключей* одинаковых элементов выбирается тот, который будет сохранен в новом *массиве*? Происходит следующее. Все элементы *массива*преобразуются в строки и *сортируются*. Затем обработчик запоминает первый *ключ* для каждого *значения*, а остальные *ключи* игнорирует.

Попробуем избавиться от повторяющихся языков в списке изученных.

<?php

$langs =

array("Lisp","Java","Python","Java",

"PHP","Perl","Lisp");

print\_r(array\_unique($langs));

?>

Получим следующее:

Array ( [0] => Lisp [1] => Java [2] => Python [3]

=> PHP [4] => Perl )

Далее рассмотрим задачу *сортировки массива*.

### Сортировка массивов

Необходимость *сортировки* данных, в том числе и данных, хранящихся в виде *массивов*, очень часто возникает при решении самых разнообразных задач. Если в языке *Си* для того, чтобы решить эту задачу, нужно написать десятки строк кода, то в *PHP* это делается одной простой командой.

##### Функция sort

Функция *sort* имеет следующий синтаксис

sort (массив [, флаги])

и *сортирует массив*, т.е. упорядочивает его *значения* по возрастанию. Эта функция удаляет все существовавшие в *массиве* *ключи*, заменяя их числовыми индексами, соответствующими новому порядку элементов. В случае успешного завершения работы она возвращает true, иначе – false.

**Пример 7.6**.Пусть у нас есть два *массива*: *цены товаров* – их названия и, наоборот, названия товаров – их цены. Упорядочим эти *массивы*по возрастанию:

<?

$items = array(10 => "хлеб", 20 => "молоко",

30 => "бутерброд");

sort($items);

// строки сортируются в алфавитном

// порядке, ключи теряются

print\_r($items);

$rev\_items = array("хлеб" => 10,

"бутерброд" => 30, "молоко" => 20);

sort($rev\_items);

// числа сортируются по возрастанию,

// ключи теряются

print\_r($rev\_items);

?>

Пример 7.6. Применение функции sort()

Получим:

Array ( [0] => бутерброд [1] =>

молоко [2] => хлеб )

Array ( [0] => 10 [1] => 20 [2] => 30 )

В качестве дополнительного аргумента может использоваться одна из следующих констант:

* *SORT\_REGULAR* – автоматический выбор метода;
* *SORT\_NUMERIC* – сравнивать элементы *массива* как числа;
* *SORT\_STRING* – сравнивать элементы *массива* как строки.

##### Функции asort, rsort, arsort

Если требуется сохранять индексы элементов *массива* после *сортировки*, то нужно использовать функцию *asort (массив [, флаги])* . Если необходимо *отсортировать массив* в обратном порядке, т.е. от наибольшего *значения* к наименьшему, то можно задействовать функцию *rsort (массив [, флаги])* . А если при этом нужно еще и сохранить *значения* *ключей*, то следует использовать функцию *arsort(массив [, флаги])* . Как вы, наверное, заметили синтаксис у этих функций абсолютно такой же, как у функции *sort* . Соответственно и *значения* флагов могут быть такими же, как у *sort* : *SORT\_REGULAR* , *SORT\_NUMERIC* , *SORT\_STRING* . Кстати говоря, флаг *SORT\_NUMERIC* появился только в *PHP4*.

<?php

$books = array("Пушкин"=>"Руслан и Людмила",

"Толстой"=>"Война и мир",

"Лермонтов"=>"Герой нашего времени");

asort($books);

// сортируем массив,

// сохраняя значения ключей

print\_r($books);

echo "<br>";

rsort($books);

// сортируем массив в обратном порядке,

// ключи будут заменены

print\_r($books);

?>

Пример 7.7. Применение функций asort, rsort, arsort

В результате работы этого скрипта получим:

Array ( [Толстой] => Война и мир

[Лермонтов] => Герой нашего времени

[Пушкин] => Руслан и Людмила )

Array ( [0] => Руслан и Людмила

[1] => Герой нашего времени

[2] => Война и мир )

**Пример 7.8**. Допустим, мы создаем каталог описаний документов. У каждого документа есть автор, название, дата публикации и краткое содержание. Мы уже не раз отображали описания, составленные из этих характеристик. Каждый раз порядок отображения этих элементов зависел от созданной нами программы. Теперь же мы хотим иметь возможность изменять порядок отображения элементов по желанию пользователя. Составим для этого следующую форму:

<form action=task.php>

<table border=1>

<tr><td>Название </td><td><input type=text

name=title size=5> </td></tr>

<tr><td>Краткое содержание </td><td><input

type=text name=description size=5>

</td></tr>

<tr><td>Автор </td><td><input type=text

name=author size=5> </td></tr>

<tr><td>Дата публикации </td><td><input

type=text name=published size=5></td></tr>

</table>

<input type=submit value="Отправить">

</form>

Пример 7.8a. Форма для примера 7.8

Будем упорядочивать данные, переданные этой формой, по убыванию их *значений*, сохраняя при этом *значения* *ключей*. Для этого удобно воспользоваться функцией *arsort()* . Поскольку нам важен только новый порядок элементов, сохраним в новом *массиве* *ключи*исходного *массива* в нужном порядке. Мы сохраняем *ключи* исходного *массива*, поскольку они являются именами элементов, из которых конструируется описание документа, а помнить их важно. Итак, получаем такой скрипт:

<?php

print\_r($\_GET); echo "<br>";

arsort ($\_GET);

// сортируем массив в обратном порядке,

// сохраняя ключи

print\_r($\_GET); echo "<br>";

$ordered\_names = array\_keys($\_GET);

// составляем новый массив

foreach($ordered\_names as $key => $val)

echo "$key :$val <br>";

// выводим элементы нового массива

?>

Пример 7.8b. Программа обработки формы из примера 7.8

#### Сортировка массива по ключам

Очевидно, что может возникнуть необходимость в *сортировке массива по значениям ключей*. Например, если у нас есть *массив* данных о книгах, как в приведенном выше примере, то вполне вероятно, что мы захотим *отсортировать* книги по именам авторов. Для этого в PHP также не нужно писать много строк кода – можно просто воспользоваться функцией *ksort()* для *сортировки* по возрастанию (прямой порядок *сортировки* ) или *krsort()* – для *сортировки* по убыванию (обратный порядок *сортировки* ). Синтаксис этих функций опять же аналогичен синтаксису функции *sort()* .

<?php

$books = array("Пушкин"=>"Руслан и Людмила",

"Толстой"=>"Война и мир",

"Лермонтов"=>"Герой нашего времени");

ksort($books);

// сортируем массив,

// сохраняя значения ключей

print\_r($books);

?>

Пример 7.9. Сортировка массива по ключам

Получим:

Array ( [Лермонтов] => Герой нашего времени

[Пушкин] => Руслан и Людмила

[Толстой] => Война и мир )

#### Сортировка с помощью функции, заданной пользователем

Кроме двух простых способов *сортировки значений массива* (по убыванию или по возрастанию) PHP предлагает пользователю возможность самому задавать критерии для *сортировки* данных. Критерий задается с помощью функции, имя которой указывается в качестве аргумента для специальных функций *сортировки* *usort()* или *uksort()* . По названиям этих функций можно догадаться, что *usort() сортируетзначения* элементов *массива*, а *uksort()* – *значения* *ключей массива* с помощью определенной пользователем функции. Обе функции возвращают true, если *сортировка* прошла успешно, и false – в противном случае. Их синтаксис выглядит следующим образом:

usort (массив , сортирующая функция)

uksort (массив , сортирующая функция)

Конечно же, нельзя *сортировать массив* с помощью любой пользовательской функции. Эта функция должна удовлетворять определенным критериям, позволяющим сравнивать элементы *массива*. Как должна быть устроена сортирующая функция? Во-первых, она должна иметь два аргумента. В них интерпретатор будет передавать пары *значений* элементов для функции *usort()* или *ключей массива* для функции *uksort()* . Во-вторых, сортирующая функция должна возвращать:

* целое число, меньшее нуля, если первый аргумент меньше второго;
* число, равное нулю, если два аргумента равны;
* число большее нуля, если первый аргумент больше второго.

Как и для других функций *сортировки*, для функции *usort()* существует аналог, не изменяющий *значения* *ключей*, – функция *uasort()* .

**Пример 7.10**. Допустим, у нас есть *массив*, содержащий такие сведения о литературных произведениях, как название, автор и год создания. Мы хотим упорядочить книги по дате создания.

<?php

// массив выглядит таким образом:

$books = array("Герой нашего времени" =>

array ("Лермонтов", 1840),

"Руслан и Людмила" => array("Пушкин",1820),

"Война и мир" => array ("Толстой",1863),

"Идиот" => array("Достоевский",1868));

/\* можно, конечно переписать этот массив

по-другому, сделав год издания, например,

индексом, но гораздо удобнее написать свою

функцию для сортировки \*/

uasort($books,"cmp");

// сортируем массив с помощью функции cmp

foreach ($books as $key => $book) {

echo "$book[0]: \"$key\"<br>";

}

function cmp($a,$b){

// функция, определяющая способ сортировки

if ($a[1] < $b[1]) return -1;

elseif ($a[1]==$b[1]) return 0;

else return 1;

}

?>

Пример 7.10. Сортировка с помощью пользовательских функций

В результате получим:

Пушкин: "Руслан и Людмила"

Лермонтов: "Герой нашего времени"

Толстой: "Война и мир"

Достоевский: "Идиот"

Мы применили нашу собственную функцию *сортировки* ко всем элементам *массива*. Далее рассмотрим, как *применить к элементам массива любую другую пользовательскую функцию*.

### Заключение

Итак, подведем итоги. В этой лекции мы изучили ряд функций, упрощающих работу с *массивами* данных. Мы рассмотрели функции для *поиска значения среди элементов массива* ; функции для *сортировки элементов массива*, как по их *значениям*, так и по *ключам* ; функции, позволяющие *применять ко всем элементам массива функцию, созданную пользователем*. Кроме того, мы изучили функцию, *выделяющую подмассивы* из элементов *массива*, и функцию, вычисляющую *сумму всех элементов массива*. Использование всех этих функций было продемонстрировано на примерах. Все функции для работы с *массивами* доступны без каких-либо конфигурационных настроек *PHP*, и пользоваться ими можно абсолютно свободно. Так что, прежде чем изобретать велосипед, загляните в руководство по языку *PHP*: – вполне возможно, что кто-то сделал это до вас.

#### Тип string (строки)

***Строка*** – это набор символов. В PHP символ – это то же самое, что байт, это значит, что существует ровно 256 различных символов. Это также означает, что PHP не имеет встроенной поддержки Unicode. В PHP практически не существует ограничений на размер *строк*, поэтому нет абсолютно никаких причин беспокоиться об их длине.

*Строка* в PHP может быть определена тремя различными *способами*:

* с помощью *одинарных кавычек* ;
* с помощью *двойных кавычек* ;
* *heredoc-синтаксисом*.

##### Одинарные кавычки

Простейший *способ* определить *строку* – это заключить ее в *одинарные кавычки* " ' ". Чтобы использовать *одинарную кавычку* внутри *строки*, как и во многих других языках, перед ней необходимо поставить символ обратной косой черты " \ ", т. е. экранировать ее. Если обратная косая черта должна идти перед *одинарной кавычкой* либо быть в конце *строки*, необходимо продублировать ее " \\' ".

Если внутри *строки*, заключенной в *одинарные кавычки*, обратный слэш " \ " встречается перед любым другим символом (отличным от " \ " и " ' " ), то он рассматривается как обычный символ и выводится, как и все остальные. Поэтому обратную косую черту необходимо экранировать, только если она находится в конце *строки*, перед закрывающей кавычкой.

В PHP существует ряд комбинаций символов, начинающихся с символа обратной косой черты. Их называют ***управляющими последовательностями***, и они имеют специальные значения, о которых мы расскажем немного позднее. Так вот, в отличие от двух других синтаксисов, *переменные* и *управляющие последовательности* для специальных символов, встречающиеся в *строках*, заключенных в *одинарные кавычки*, не *обрабатываются* .

<?php

echo 'Также вы можете вставлять в строки

символ новой строки таким образом,

поскольку это нормально';

// Выведет: Чтобы вывести ' надо

// перед ней поставить \

echo 'Чтобы вывести \' надо перед ' .

'ней поставить \\';

// Выведет: Вы хотите удалить C:\\*.\*?

echo 'Вы хотите удалить C:\\\*.\*?';

// Выведет: Это не вставит: \n новую строку

echo 'Это не вставит: \n новую строку';

// Выведет: Переменные $expand также

// $either не подставляются

echo 'Переменные $expand также $either' .

'не подставляются';

?>

Пример 2.7. Использование управляющих последовательностей

##### Двойные кавычки

Если *строка* заключена в *двойные кавычки* " " ", PHP распознает большее количество *управляющих последовательностей* для специальных символов. Некоторые из них приведены в [таблице 2.7](http://www.intuit.ru/studies/courses/42/42/lecture/27177?page=5#table.2.7).

|  |  |
| --- | --- |
| Таблица 2.7. *Управляющие последовательности* | |
| **Последовательность** | **Значение** |
| \n | Новая *строка* ( LF или 0x0A (10) в ASCII) |
| \r | Возврат каретки ( CR или 0x0D (13) в ASCII) |
| \t | Горизонтальная табуляция ( HT или 0x09 (9) в ASCII) |
| \\ | Обратная косая черта |
| \$ | Знак доллара |
| \" | Двойная кавычка |

Повторяем, если вы захотите экранировать любой другой символ, обратная косая черта также будет напечатана!

Самым важным свойством *строк* в *двойных кавычках* является *обработка переменных*.

##### Heredoc

Другой *способ определения строк* – это использование ***heredoc-синтаксиса***. В этом случае *строка* должна начинаться с символа <<<, после которого идет идентификатор. Заканчивается *строка* этим же идентификатором. Закрывающий идентификатор должен начинаться в первом столбце *строки*. Кроме того, идентификатор должен соответствовать тем же правилам именования, что и все остальные метки в PHP: содержать только буквенно-цифровые символы и знак подчеркивания и начинаться не с цифры или знака подчеркивания.

*Heredoc* -текст ведет себя так же, как и *строка* в *двойных кавычках*, при этом их не имея. Это означает, что вам нет необходимости экранировать кавычки в *heredoc*, но вы по-прежнему можете использовать перечисленные выше *управляющие последовательности*. *Переменные* внутри *heredoc* тоже *обрабатываются*.

<?php

$str = <<<EOD

Пример строки, охватывающей несколько

строчек, с использованием

heredoc-синтаксиса

EOD;

// Здесь идентификатор – EOD. Ниже

// идентификатор EOD

$name = 'Вася';

echo <<<EOD

Меня зовут "$name".

EOD;

// это выведет: Меня зовут "Вася".

?>

Пример 2.8. Использование heredoc-синтаксиса

Замечание: Поддержка *heredoc* была добавлена в PHP 4.

### Поиск элемента в строке

Для того чтобы определить, входит ли данная *подстрока* в состав *строки*, используется *функция* *strpos()* . *Синтаксис* *strpos()* такой:

strpos (исходная строка,строка для поиска

[,с какого символа искать])

Она возвращает позицию появления искомой *строки* в исходной *строке* или возвращает логическое false, если вхождение не найдено. Дополнительный *аргумент* позволяет задавать символ, начиная с которого будет производиться *поиск*. Кроме логического false эта *функция* может возвращать и другие значения, которые приводятся к false (например, 0 или ""). Поэтому для того, чтобы проверить, найдена ли искомая *строка*, рекомендуют использовать оператор эквивалентности " === ".

<?

$str = "Идея наносить данные на перфокарты

и затем считывать и обрабатывать их

автоматически принадлежала Джону Биллингсу,

а ее техническое решение осуществил Герман

Холлерит. Перфокарта Холлерита оказалась

настолько удачной, что без малейших изменений

просуществовала до наших дней.";

$pos = strpos($str,"Холлерит");

if ($pos !== false) echo "Искомая строка

встречена в позиции номер $pos ";

else echo "Искомая строка не найдена";

/\* заметим, что мы проверяем значение

$pos на эквивалентность с false.

Иначе строка, находящаяся в первой позиции,

не была бы найдена, так как 0

интерпретируется как false. \*/

?>

Пример 8.3. Использование функции strpos()

Если *значение* параметра строка\_для\_поиска не является *строкой*, то оно преобразуется к целому типу и рассматривается как ASCII-код символа. Чтобы получить ASCII-код любого символа в *PHP*, можно воспользоваться функцией *ord("символ")*

Например, если мы напишем $pos = *strpos*($str,228); то *интерпретатор* будет считать, что мы ищем символ " д ". Если добавить эту строчку в приведенный выше пример и вывести результат, то получим сообщение, что искомая *строка* найдена в позиции 1.

*Функция*, обратная по смыслу ord, – это *chr (код символа)* . Она по ASCII-коду выводит символ, соответствующий этому *коду*.

С помощью функции *strpos* можно найти номер только первого появления *строки* в исходной *строке*. Естественно, есть функции, которые позволяют вычислить номер последнего появления *строки* в исходной *строке*. Это *функция* *strrpos()* . Ее *синтаксис* таков:

strrpos (исходная строка, символ для поиска)

В отличие от *strpos()* эта *функция* позволяет найти позицию последнего появления в *строке* указанного символа.

Бывают ситуации, когда знать позицию, где находится искомая *строка*, необязательно, а нужно просто получить все символы, которые расположены после вхождения этой *строки*. Можно, конечно, воспользоваться и приведенными выше функциями *strpos()* и *strrpos()*, но можно сделать и проще – *выделить подстроку* с помощью предназначенных именно для этого функций.

### Выделение подстроки

#### Функция strstr

Говоря о *выделении подстроки* из искомой *строки* в языке PHP, в первую очередь стоит отметить функцию *strstr()* :

strstr (исходная строка, строка для поиска)

Она находит первое появление искомой *строки* и возвращает подстроку, начиная с этой искомой *строки* до конца исходной *строки*.

Если *строка* для поиска не найдена, то функция вернет false. Если *строка* для поиска не принадлежит *строковому типу* данных, то она переводится в целое число и рассматривается как *код символа*. Кроме того, эта функция *чувствительна к регистру*, т.е. если мы будем параллельно искать вхождения слов "Идея" и "идея", то результаты будут разными. Вместо *strstr()* можно использовать абсолютно идентичную ей функцию *strchr()* .

**Пример 8.4**. Выделим из *строки*, содержащей название и автора исследования, подстроку, начинающуюся со слова "Название":

<?

$str = "Автор: Иванов Иван (<a

href=mailto:van@mail.ru>написать письмо</a>),

Название: 'Исследование языков

программирования' ";

echo "<b>Исходная строка: </b>",$str;

if (!strstr($str, "Название"))

echo "Строка не найдена<br>";

else echo "<p><b>Полученная подстрока: </b>",

strstr($str, "Название");

?>

Пример 8.4. Использование функции strstr()

В результате получим:

Исходная строка: Автор: Иванов Иван

(написать письмо),

Название: 'Исследование языков

программирования'

Полученная подстрока: Название:

'Исследование языков программирования'

Для реализации регистронезависимого поиска подстроки существует соответствующий аналог этой функции – функция *stristr (исходная строка, искомая строка)* . Действует и используется она точно так же, как и *strstr()* , за исключением того, что регистр, в котором записаны символы искомой *строки*, не играет роли при поиске.

Очевидно, что функция *strstr()* не слишком часто используется – на практике редко бывает нужно получить подстроку, начинающуюся с определенного слова или *строки*. Но в некоторых случаях и она может пригодиться. Кроме того, в PHP есть и более удобные функции для поиска вхождений. Наиболее мощные из них, конечно, связаны с регулярными выражениями. Их мы рассмотрим в одной из последующих лекций.

#### Функция substr

Иногда мы не знаем, с каких символов начинается искомая *строка*, но знаем, например, что начинается она с пятого символа и заканчивается за два символа до конца исходной *строки*. Как *выделить подстроку* по такому описанию? Очень просто, с помощью функции *substr()* . Ее синтаксис можно записать следующим образом:

substr (исходная строка,

позиция начального символа [, длина])

Эта функция возвращает часть *строки* длиной, заданной параметром длина, начиная с символа, указанного параметром позиция начального символа. Позиция, с которой начинается выделяемая подстрока, может быть как положительным целым числом, так и отрицательным. В последнем случае отсчет элементов производится с конца *строки*. Если параметр длина опущен, то *substr()*возвращает подстроку от указанного символа и до конца исходной *строки*. Длина выделяемой подстроки тоже может быть задана отрицательным числом. Это означает, что указанное число символов отбрасывается с конца *строки*.

**Пример 8.5**. Допустим, у нас есть фраза, выделенная жирным шрифтом с помощью тега <b> языка HTML. Мы хотим получить эту же фразу, но в обычном стиле. Напишем такую программу:

<?php

$word = "<b>Hello, world!</b>";

echo $word , "<br>";

$pure\_str = substr($word, 3, -4);

/\* выделяем подстроку,

начиная с 3-го символа,

не включая 4 символа с конца строки \*/

echo $pure\_str;

?>

Пример 8.5. Использование функции substr()

В результате работы этого скрипта получим:

Hello, world!

Hello, world!

На самом деле решить такую задачу можно гораздо проще, с помощью функции *strip\_tags* :

strip\_tags (строка [, допустимые теги])

Эта функция возвращает *строку*, из которой удалены все html и php-теги. С помощью дополнительного аргумента можно задать теги, которые не будут удалены из *строки*. Список из нескольких тегов вводится без каких-либо знаков разделителей. Функция выдает предупреждение, если встречает неправильные или неполные теги.

<?php

$string = "<b>Bold text</b>

<i>Italic text</i>";

$str = strip\_tags($string);

// удаляем все теги из строки

$str1 = strip\_tags($string, '<b>');

// удаляем все теги кроме тега <b>

$str2 = strip\_tags($string, '<i>');

// удаляем все теги кроме тегов <i>

echo $str,"<br>",$str1,"<br>", $str2;

?>

Пример 8.6. Использование функции strip\_tags()

В результате получим:

Bold text Italic text

**Bold text Italic text**

*Bold text Italic text*

Приведем другой пример использования функции *substr()* . Допустим, у нас есть какое-то сообщение с приветствием и подписью автора. Мы хотим удалить сначала приветствие, а потом и подпись, оставив только содержательную часть сообщения.

<?php

$text = "Привет! Сегодня мы изучаем работу

со строками. Автор.";

$no\_hello = substr($text, 8);

// убираем приветствие

$content = substr($text, 8, 38);

// то же самое, что substr($text, 8, -6).

// Убираем подпись.

echo $text, "<br>", $no\_hello,

"<br>", $content;

?>

В результате получим:

Привет! Сегодня мы изучаем работу

со строками. Автор.

Сегодня мы изучаем работу со строками. Автор.

Сегодня мы изучаем работу со строками.

Если нам нужно получить один конкретный символ из *строки*, зная его порядковый номер, то не следует задействовать функции типа *substr* . Можно воспользоваться более простым синтаксисом – записывая номер символа в фигурных скобках после имени строковой переменной. В контексте предыдущего примера букву " р ", расположенную второй по счету, можно получить так:

echo $text{1}; // выведет символ "р"

Заметим, что номером этого символа является число один, а не два, так как нумерация символов *строки* производится начиная с нуля.

Раз уж мы начали говорить о символах в *строке* и их нумерации, то невольно возникает вопрос, сколько всего символов в *строке* и как это вычислить. Число символов в *строке* – это ***длина строки*** . Вычислить *длину строки* можно с помощью функции *strlen (строка)* . Например, *длина строки* "Разработка информационной модели" вычисляется с помощью команды: *strlen* ("Разработка информационной модели"); и равна 32 символам.

Итак, как выделять и находить подстроки, мы рассмотрели. Теперь научимся заменять *строку*, входящую в состав исходной *строки*, на другую *строку* по нашему выбору.

### Замена вхождения подстроки

#### Функция str\_replace

Для *замены вхождения* подстроки можно использовать функцию *str\_replace()* . Это простая и удобная функция, позволяющая решать множество задач, не требующих особых тонкостей при выборе заменяемой подстроки. Для того чтобы производить *замены* с более сложными условиями, используют механизм регулярных выражений и соответствующие функции ereg\_replace() и preg\_replace(). Синтаксис функции *str\_replace()* такой:

str\_replace(искомое значение,

значение для замены, объект)

Функция *str\_replace()* ищет в рассматриваемом объекте значение и *заменяет* его значением, предназначенным для *замены*. Почему мы говорим здесь не про *строки* для поиска и замены и исходную *строку*, а про значения и объект, в котором происходит *замена*? Дело в том, что начиная с PHP 4.0.5 любой аргумент этой функции может быть массивом.

Если объект, в котором производится поиск и *замена*, является массивом, то эти действия выполняются для каждого элемента массива и в результате возвращается новый массив.

<?php

$greeting = array("Привет", "Привет всем!",

"Привет, дорогая!"); // объект

$new\_greet = str\_replace("Привет",

"Доброе утро", $greeting);

// делаем замену

print\_r($new\_greet);

/\* получим: Array ([0]=>Доброе утро

[1]=>Доброе утро всем!

[2]=>Доброе утро, дорогая!) \*/

?>

Пример 8.7. Использование функции str\_replace()

Если искомое значение и значение для *замены* – массивы, то берется по одному значению из каждого массива и производится их поиск и *замена* в объекте. Если значений для *замены* меньше, чем значений для поиска, то в качестве новых значений используется пустая *строка*.

<?php

$greeting = array("Привет", "Привет всем!",

"Привет, дорогая!","Здравствуйте",

"Здравствуйте, товарищи", "Hi");

// объект

$search = array ("Привет",

"Здравствуйте", "Hi");

// значения, которые будем заменять

$replace = array ("Доброе утро",

"День добрый");

// значения, которыми будем заменять

$new\_greet = str\_replace($search, $replace,

$greeting);

// делаем замену

print\_r($new\_greet);

//выводим полученный массив

?>

Пример 8.8. Использование функции str\_replace(). Вариант 2

В результате получим такой массив:

Array (

[0] => Доброе утро

[1] => Доброе утро всем!

[2] => Доброе утро, дорогая!

[3] => День добрый

[4] => День добрый, товарищи

[5] =>

)

Если значения для поиска – массив, а значение для *замены* – *строка*, то эта *строка* будет использована для *замены* всех найденных значений.

<?php

$greeting = array("Привет", "Привет всем!",

"Привет, дорогая!", "Здравствуйте",

"Здравствуйте, товарищи");

// объект

$search = array ("Привет","Здравствуйте");

// значения, которые будем заменять

$replace = "День добрый";

// значение, которым будем заменять

$new\_greet = str\_replace($search,

$replace, $greeting); // делаем замену

print\_r($new\_greet);

//выводим полученный массив

?>

Пример 8.9. Использование функции str\_replace(). Вариант 3

Получим:

Array (

[0] => День добрый

[1] => День добрый всем!

[2] => День добрый, дорогая!

[3] => День добрый

[4] => День добрый, товарищи

)

Функция *str\_replace()* *чувствительна к регистру*, но существует ее регистронезависимый аналог – функция *str\_ireplace()* . Однако эта функция поддерживается не во всех версиях PHP.

#### Функция substr\_replace

Эта функция сочетает в себе свойства двух уже рассмотренных нами функций – функции *str\_replace()* и *substr()* . Ее синтаксис таков:

substr\_replace (исходная строка,

строка для замены,

позиция начального символа [, длина])

Эта функция *заменяет часть строки строкой, предназначенной для замены*. *Заменяется* та часть *строки* (т.е. подстрока), которая начинается с позиции, указанной параметром позиция начального символа. С помощью дополнительного аргумента длина можно ограничить число заменяемых символов. То есть, фактически, мы не указываем конкретно *строку*, которую нужно *заменить*, мы только описываем, где она находится и, возможно, какую длину имеет. В этом отличие функции *substr\_replace()* от *str\_replace()* .

Как и в случае с функцией *substr()* аргументы позиция начального символа и длина могут быть отрицательными. Если позиция начального символа отрицательна, то *замена* производится, начиная с этой позиции относительно конца *строки*. Отрицательная длина задает, сколько символов от конца *строки* не должно быть заменено. Если длина не указывается, то *замена* происходит до конца *строки*.

<?php

$text = "Меня зовут Вася.";

echo "Исходная строка: $text<hr>\n";

/\* Следующие две строки заменят всю

исходную строку строкой 'А меня – Петя' \*/

echo substr\_replace($text, 'А меня – Петя',

0) . "<br>\n";

echo substr\_replace($text, 'А меня – Петя',

0, strlen($text)) . "<br>\n";

// Следующая строка добавит слово 'Привет! '

// в начало исходной строки

echo substr\_replace($text, 'Привет! ',

0, 0) . "<br>\n";

// Следующие две строки заменят имя Вася

// на имя Иван в исходной строке

echo substr\_replace($text, 'Иван', 11,

-1) . "<br>\n";

echo substr\_replace($text, 'Иван', -5,

-1) . "<br>\n";

?>

Пример 8.10. Использование функции substr\_replace()

В результате работы этого скрипта получим:

Исходная строка: Меня зовут Вася.

------------------------------------------

А меня – Петя

А меня – Петя

Привет! Меня зовут Вася.

Меня зовут Иван.

Меня зовут Иван.

### Разделение и соединение строки

Очень полезные функции – *функция* *разделения строки на части* и обратная ей *функция* *объединения строк* в одну *строку*. Почему очень полезные? Например, если вы динамически генерируете форму по желанию пользователя, можно предложить ему вводить элементы для создания списка выбора, разделяя их каким-нибудь символом. И для того чтобы обработать полученный *список* значений, как раз и пригодится умение *разбивать строку* на кусочки. Для реализации такого разбиения в *PHP* можно использовать несколько функций:

explode(разделитель,исходная строка

[,максимальное число элементов])

split (шаблон, исходная строка

[, максимальное число элементов])

preg\_split (шаблон, исходная строка

[, максимальное число элементов

[,флаги]])

Последние две функции работают с регулярными выражениями, поэтому в данной лекции мы их рассматривать не будем. Рассмотрим более простую функцию – *explode()* .

*Функция* *explode()* *делит исходную строку на подстроки*, каждая из которых отделена от соседней с помощью указанного разделителя, и возвращает *массив* полученных *строк*. Если задан дополнительный *параметр* максимальное число элементов, то *число элементов в массиве* будет не больше этого параметра, в последний элемент записывается весь *остаток* *строки*. Если в качестве разделителя указана пустая *строка* """", то *функция* *explode()* вернет false. Если символа разделителя в исходной *строке* нет, то возвращается *массив* с исходной *строкой* без изменений.

**Пример 8.11**. мы хотим создать элемент формы – выпадающий *список* и значения для этого списка должен ввести *пользователь*, не знакомый с языком *html*. Создадим такую форму:

<form action=exp.php>

Введите варианты для выбора автора статьи

через двоеточие (":"):<br>

<input type=text name=author size=40>

<br>

<input type=submit value="Создать элемент">

</form>

Пример 8.11. Использование функции explode()

*Скрипт*, который будет ее обрабатывать (*exp*.*php*), может быть таким:

<?php

$str = $\_GET["author"];

$names = explode(":",$str);

// разбиваем строку введенную,

// пользователем с помощью ":"

$s = "<select name=author>";

// создаем выпадающий список

foreach ($names as $k => $name) {

$s .= "<option value=$k>$name";

// добавляем элементы к списку

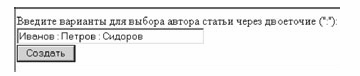
}

$s .= "</select>";

echo $s;

?>

В итоге, если мы введем такую строчку в форму:



**Рис. 8.2.** Ввод значений для создания выпадающего списка

то получим следующий выпадающий *список*:



**Рис. 8.3.** Выпадающий список, полученный в результате обработки формы

Кроме *разделения строки на части* иногда, наоборот, возникает необходимость *объединения нескольких строк в одно целое*. *Функция*, предлагаемая для этого языком *PHP*, называется *implode()* :

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

Эта *функция* объединяет элементы массива с помощью переданного ей объединяющего элемента (например, запятой). В отличие от функции *explode()* , порядок аргументов в функции *implode()* не имеет значения.

**Пример 8.12**. Допустим, мы храним имя, фамилию и отчество человека по отдельности, а выводить их на странице нужно вместе. Чтобы *соединить их в одну строку*, можно использовать функцию *implode()* :

<?php

$data = array("Иванов","Иван","Иванович");

$str = implode(" ", $data);

echo $str;

?>

Пример 8.12. Использование функции implode()

В результате работы этого скрипта получим *строку*:

Иванов Иван Иванович

У функции *implode()* существует *псевдоним* – *функция* join(), т.е. эти две функции отличаются лишь именами.

### Заключение

Итак, мы завершили знакомство с функциями работы со *строками* языка *PHP*. Конечно же, мы затронули далеко не все существующие функции, а лишь малую часть. Мы изучили функции, позволяющие *найти набор символов в строке*, функции, *заменяющие все вхождения одной строки на другую*, функции *разделения строки на части* и *соединения нескольких строк в одну*, а также рассмотрели функции, позволяющие *выводить на экран строки, содержащие html–код* без их форматирования браузером.

**4.4 Стандартные функции PHP для работы с файловой структурой и управления временем. Управление сессиями**

### Создание файла

#### Функция fopen

Вообще говоря, в PHP не существует функции, предназначенной именно для *создания файлов.* Большинство функций работают с уже существующими файлами в файловой системе сервера. Есть несколько функций, которые позволяют создавать временные файлы, или, что то же самое, файлы с уникальным для *текущей директории* именем. А вот для того, чтобы создать самый обычный файл, нужно воспользоваться функцией, которая открывает локальный или удаленный файл. Называется эта функция *fopen*(). Что значит "открывает файл"? Это значит, что *fopen* связывает данный файл с *потоком управления* программы. Причем связывание бывает различным в зависимости от того, что мы хотим делать с этим файлом: читать его, *записывать* в него данные или делать и то и другое. Синтаксис этой функции такой:

resource fopen ( имя\_файла, тип\_доступа

[, use\_include\_path])

В результате работы эта функция возвращает указатель (типа ресурс) на открытый ею файл. В качестве параметров этой функции передаются: имя файла, который нужно открыть, *тип доступа к файлу* (определяется тем, что мы собираемся делать с ним) и, возможно, параметр, определяющий, искать ли указанный файл в *include\_path*. Есть еще один опциональный параметр, но о нем мы говорить не будем, дабы не усложнять изложение. Обсудим подробнее каждый из этих трех параметров.

Параметр имя\_файла должен быть строкой, содержащей правильное локальное имя файла или URL-адрес файла в сети. Если имя файла начинается с указания протокола доступа (например, http://... или ftp://...), то интерпретатор считает это имя адресом URL и ищет обработчик указанного в URL протокола. Если обработчик найден, то PHP проверяет, разрешено ли работать с объектами URL как с обычными файлами (директива allow\_url\_fopen ). Если allow\_url\_fopen=off, то функция *fopen* вызывает ошибку и генерируется предупреждение. Если имя файла не начинается с протокола, то считается, что указано имя локального файла. Чтобы открыть локальный файл, нужно, чтобы PHP имел соответствующие права доступа к этому файлу.

Параметр use\_include\_path, установленный в значение 1 или TRUE, заставляет интерпретатор искать указанный в *fopen*() файл в *include\_path*. Напомним, что *include\_path* - это директива из *файла настроек* PHP, задающая список директорий, в которых могут находиться файлы для включения. Кроме функции *fopen*() она используется функциями include() и require().

Параметр тип\_доступа может принимать одно из следующих значений (см. [таб. 9.1](http://www.intuit.ru/studies/courses/42/42/lecture/27191?page=1#table.9.1)).

Итак, чтобы создать файл, нужно, как бы нелепо это ни звучало, открыть несуществующий файл на запись.

<?php

$h = fopen("my\_file.html","w");

/\* открывает на запись файл my\_file.html,

если он существует, или создает пустой

файл с таким именем, если его еще нет \*/

$h = fopen("dir/another\_file.txt","w+");

/\* открывает на запись и чтение или создает

файл another\_file.txt в директории dir \*/

$h = fopen(

"http://www.server.ru/dir/file.php","r");

/\* открывает на чтение файл, находящийся по

указанному адресу\*/

?>

Пример 9.1. Использование функции fopen()

Создавая файл, нужно учитывать, под какой операционной системой вы работаете, и под какой ОС предположительно этот файл будет читаться. Дело в том, что разные операционные системы по-разному отмечают конец строки. В Unix-подобных ОС конец строки обозначается \n, в системах типа Windows - \r\n. Windows предлагает специальный флаг t для перевода *символов конца строки* систем типа Unix в свои *символы конца строки*. В противоположность этому существует флаг b, используемый чаще всего для *бинарных файлов*, благодаря которому такой трансляции не происходит. Использовать эти флаги можно, просто дописав их после последнего символа выбранного *типа доступа к файлу.* Например, открывая файл на чтение, вместо r следует использовать rt, чтобы перекодировать все *символы конца строки* в \r\n. Если не использовать флаг b при открытии *бинарных файлов*, то могут появляться ошибки, связанные с изменением содержимого файла. Из соображений переносимости программы на различные платформы рекомендуется всегда использовать флаг b при открытии файлов с помощью *fopen*().

|  |  |
| --- | --- |
| Таблица 9.1. Значения принимаемые параметром тип доступа | |
| **Тип доступа** | **Описание** |
| r | Открывает файл только для чтения; устанавливает указатель позиции в файле на начало файла. |
| r+ | Открывает файл для чтения и записи; устанавливает указатель файла на его начало. |
| w | Открывает файл только для записи; устанавливает указатель файла на его начало и усекает файл до нулевой длины. Если файл не существует, то пытается создать его. |
| w+ | Открывает файл для чтения и записи; устанавливает указатель файла на его начало и усекает файл до нулевой длины. Если файл не существует, то пытается создать его. |
| a | Открывает файл только для записи; устанавливает указатель файла в его конец. Если файл не существует, то пытается создать его. |
| a+ | Открывает файл для чтения и записи; устанавливает указатель файла в его конец. Если файл не существует, то пытается создать его. |
| x | Создает и открывает файл только для записи; помещает указатель файла на его начало. Если файл уже существует, то *fopen*()возвращает false и генерируется предупреждение. Если файл не существует, то делается попытка создать его. Этот тип доступа поддерживается начиная с версии PHP 4.3.2 и работает только с локальными файлами. |
| x+ | Создает и открывает файл для чтения и записи; помещает указатель файла на его начало. Если файл уже существует, то *fopen*() возвращает false и генерируется предупреждение. Если файл не существует, то делается попытка создать его. Этот тип доступа поддерживается, начиная с версии PHP 4.3.2, и работает только с локальными файлами. |

Что происходит, если открыть или создать файл с помощью *fopen* не удается? В этом случае PHP генерирует предупреждение, а функция *fopen* возвращает как результат своей работы значение false. Такого рода предупреждения можно "подавить" (запретить) с помощью символа *@*.

Например, такая команда не выведет предупреждения, даже если открыть файл не удалось:

$h = @fopen("dir/another\_file.txt","w+");

Таким образом, функция *fopen*() позволяет создать только лишь пустой файл и сделать его *доступным для записи*. Как же *записать данные* в этот файл? Как прочитать данные из уже существующего файла?

Прежде чем ответить на эти вопросы, рассмотрим, как *закрыть* установленное с помощью *fopen*() соединение.

### Закрытие соединения с файлом

После выполнения необходимых действий с файлом, будь то *чтение* или *запись данных* или что-либо другое, соединение, установленное с этим файлом функцией *fopen*(), нужно *закрыть.* Для этого используют функцию fclose(). *Синтаксис* у нее следующий:

fclose (указатель на файл)

Эта *функция* возвращает TRUE, если соединение успешно *закрыто*, и FALSE - в противном случае. *Параметр* этой функции должен указывать на *файл*, успешно открытый, например, с помощью функции *fopen*().

<?php

$h = fopen("my\_file.html","w");

fclose($h);

?>

Пример 9.2. Использование функции fclose()

Конечно, если не закрывать соединение с файлом, никаких ошибок выполнения скрипта не произойдет. Но в целом для сервера это может иметь серьезные последствия. Например, *хакер* может воспользоваться открытым соединением и записать в *файл* *вирус*, не говоря уже о лишней трате ресурсов сервера. Так что советуем всегда закрывать соединение с файлом после выполнения необходимых действий.

### Запись данных в файл

#### Функция fwrite

Для того чтобы *записать данные в файл*, доступ к которому открыт функцией *fopen*(), можно использовать функцию fwrite(). Синтаксис у нее следующий:

int fwrite ( указатель на файл,

строка [, длина])

Эта функция записывает содержимое строки в файл, на который указывает указатель на файл. Если указан дополнительный аргумент, то запись заканчивается после того, как записано количество символов, равное значению этого аргумента, или когда будет достигнут конец строки.

В результате своей работы функция fwrite() возвращает число записанных байтов или false, в случае ошибки.

**Пример 9.3**. Пусть в нашей рабочей директории нет файла my\_file.html. Создадим его и запишем в него строку текста:

<?php

$h = fopen("my\_file.html","w");

$text = "Этот текст запишем в файл.";

if (fwrite($h,$text))

echo "Запись прошла успешно";

else

echo "Произошла ошибка при записи данных";

fclose($h);

?>

Пример 9.3. Использование функции fwrite()

В результате работы этого скрипта в браузере мы увидим сообщение о том, что запись прошла успешно, а в файле my\_file.html появится строка "Этот текст запишем в файл.". Если бы этот *файл существовал* до того, как мы выполнили этот скрипт, все находящиеся в нем данные были бы удалены.

Если же мы напишем такой скрипт:

<?php

$h = fopen("my\_file.html","a");

$add\_text = "Добавим текст в файл.";

if(fwrite($h,$add\_text,7))

echo "Добавление текста прошло

успешно<br>";

else echo "Произошла ошибка при

добавлении данных<br>";

fclose($h);

?>

то к строке, уже существующей в файле my\_file.html, добавится еще семь символов из строки, содержащейся в переменной $add\_text, т.е. слово "Добавим"

Функция fwrite() имеет псевдоним fputs(), используемый таким же образом, что и сама функция.

Далее мы рассмотрим, какие методы *чтения данных из файла* предлагает язык PHP.

### Чтение данных из файла

Если мы хотим прочитать данные из существующего файла, одной функции *fopen*(), как и в случае с *записью данных*, недостаточно. Она лишь возвращает *указатель* на открытый *файл*, но не *считывает* ни одной строки из этого файла. Поэтому для того, чтобы прочитать данные из файла, нужно воспользоваться одной из специальных функций: file, *readfile*, *file\_get\_contents*, *fread*, fgets и т.п.

#### Функция fread

Эта функция осуществляет *чтение данных из файла*. Ее можно использовать и для *чтения данных* из *бинарных файлов*, не опасаясь их повреждения. Синтаксис *fread*() такой:

string fread (указатель на файл, длина)

При вызове этой функции происходит *чтение данных* длины (в байтах), определенной параметром длина, из файла, на который указывает указатель на файл. Параметр указатель на файл должен быть реально существующей переменной типа ресурс, содержащей в себе связь с файлом, открытую, например, с помощью функции *fopen*(). Чтение данных происходит до тех пор, пока не встретится конец файла или пока не будет прочитано указанное параметром длина число байтов.

В результате работы функция *fread*() возвращает строку со *считанной* из файла информацией.

Как вы заметили, в этой функции параметр длина - обязательный. Следовательно, если мы хотим *считать* весь файл в строку, нужно знать его длину. PHP может самостоятельно вычислить длину указанного файла. Для этого нужно воспользоваться функцией filesize(имя файла). В случае ошибки эта функция вернет false. К сожалению, ее можно использовать только для получения *размера локальных файлов*.

**Пример 9.4**. Прочитаем содержимое файла my\_file.html

<?php

$h = fopen("my\_file.html","r+");

// открываем файл на запись и чтение

$content = fread($h,

filesize("my\_file.html"));

// считываем содержимое файла в строку

fclose($h); // закрываем соединение с файлом

echo $content;

// выводим содержимое файла

// на экран браузера

?>

Пример 9.4. Использование функции fread()

Для того чтобы *считать* содержимое *бинарного файла*, например изображения, в таких системах, как Windows, рекомендуется открывать файл с помощью флага rb или ему подобных, содержащих символ b в конце.

Функция filesize() кэширует результаты своей работы. Если изменить содержимое файла my\_file.html и снова запустить приведенный выше скрипт, то результат его работы не изменится. Более того, если запустить скрипт, считывающий данные из этого файла с помощью другой функции (например, fgets ), то результат может оказаться таким, как если бы файл не изменился. Чтобы этого избежать, нужно очистить статический кэш, добавив в код программы команду clearstatcache() ;

#### Функция fgets

С помощью функции fgets() можно *считать из файла строку* текста. Синтаксис этой функции практически такой же, как и у *fread*(), за исключением того, что длину *считываемой строки* указывать необязательно:

string fgets ( указатель на файл, [ длина])

В результате работы функция fgets() возвращает строку длиной ( длина-1 ) байт из файла, на который указывает указатель на файл. Чтение заканчивается, если прочитано ( длина-1 ) символов или встретился символ перевода строки или конец файла. Напомним, что в PHP один символ - это один байт. Если длина *считываемой строки* не указана (данная возможность появилась начиная с PHP 4.2.0), то считывается 1 Кбайт (1024 байт) текста или, что то же самое, 1024 символа. Начиная с версии PHP 4.3, если параметр длина не задан, *считывается строка* целиком. В случае ошибки функция fgets() возвращает false. Для версий PHP начиная с 4.3 эта *функция безопасна* для двоичных файлов.

<?php

$h = fopen("my\_file.html","r+");

$content = fgets($h,2);

// считает первый символ из

// первой строки файла my\_file.html

fclose($h);

echo $content;

?>

Пример 9.5. Использование функции fgets()

Обе функции, *fread*() и fgets(), прекращают считывание данных из файла, если встречают конец файла. В PHP есть специальная функция, проверяющая, смотрит ли указатель позиции файла на конец файла. Это булева функция feof(), в качестве параметра которой передается указатель на соединение с файлом.

Например, вот так можно *считать* все строки файла my\_file.html:

<?php

$h = fopen("my\_file.html","r");

while (!feof ($h)) {

$content = fgets($h);

echo $content,"<br>";

}

fclose($h);

?>

#### Функция fgetss

Существует разновидность функции fgets() - функция fgetss(). Она тоже позволяет *считывать* строку из указанного файла, но при этом удаляет из него все встретившиеся html-теги, за исключением, быть может, некоторых. Синтаксис fgetss() такой:

string fgetss(указатель на файл,

длина [, допустимые теги])

Обратите внимание, что здесь аргумент длина обязательный.

**Пример 9.6**. Пусть у нас имеется файл my\_file.html следующего содержания:

<h1>Без труда не вынешь и рыбку из пруда.</h1>

<b>Тише едешь - дальше будешь</b> У семи нянек<i> дитя без глазу</i>.

Выведем на экран все строки файла my\_file.html, удалив из них все теги, кроме <b> и <i>:

<?php

$h = fopen("my\_file.html","r");

while (!feof ($h)) {

$content = fgetss($h,1024,'<b><i>');

echo $content,"<br>";

}

fclose($h);

?>

Пример 9.6. Использование функции fgetss()

В результате работы этого скрипта получим:

Без труда не вынешь и рыбку из пруда. Тише едешь - дальше будешь У семи нянек дитя без глазу.

#### Функция fgetc

Естественно, если можно считывать информацию из файла построчно, то можно *считывать* ее и посимвольно. Для этого предназначена функция fgetc(). Легко догадаться, что синтаксис у нее следующий:

string fgetc ( указатель на файл )

Эта функция возвращает символ из файла, на который ссылается указатель на файл, и значение, вычисляемое как FALSE, если встречен конец строки.

Вот так, например, можно *считать* файл по одному символу:

<?php

$h = fopen("my\_file.html","r");

while (!feof ($h)) {

$content = fgetc($h);

echo $content,"<br>";

}

fclose($h);

?>

На самом деле для того чтобы прочитать содержимое файла, открывать соединение с ним посредством функции *fopen*() совсем не обязательно. В PHP есть функции, которые позволяют делать это, используя лишь имя файла. Это функции *readfile*( ), file( ) и *file\_get\_contents*( ). Рассмотрим каждую из них подробнее.

#### Функция readfile

Синтаксис:

int readfile ( имя\_файла

[, use\_include\_path])

Функция *readfile*() считывает файл, имя которого передано ей в качестве параметра имя\_файла, и выводит его содержимое на экран. Если дополнительный аргумент use\_include\_path имеет значение TRUE, то поиск файла с заданным именем производится и по директориям, входящим в *include\_path*.

В программу эта функция возвращает число *считанных* байтов (символов) файла, а в случае ошибки - FALSE. Сообщения об ошибке в этой функции можно подавить оператором *@*.

**Пример 9.7**. Следующий скрипт выведет на экран содержимое файла my\_file1.html и *размер этого файла*, если он существует. В противном случае выведется наше сообщение об ошибке - строка "Error in *readfile*".

<?php

$n = @readfile ("my\_file1.html");

/\* выводит на экран содержимое файла и

записывает его размер в переменную $n \*/

if (!$n) echo "Error in readfile";

/\* если функция readfile() выполнилась

с ошибкой, то $n=false и выводим

сообщение об ошибке \*/

else echo $n;

// если ошибки не было, то выводим число

// считанных символов

?>

Пример 9.7. Использование функции readfile()

С помощью функции *readfile*() можно читать содержимое удаленных файлов, указывая их URL-адрес в качестве имени файла, если эта опция не отключена в настройках сервера.

Сразу же выводить содержимое файла на экран не всегда удобно. Порой нужно записать информацию из файла в переменную, чтобы в дальнейшем произвести с ней какие-либо действия. Для этого можно использовать функцию file() или *file\_get\_contents*().

#### Функция file

Функция *file()* предназначена для считывания информации из файла в переменную типа *массив*. Синтаксис у нее такой же, как и у функции *readfile*(), за исключением того, что в результате работы она возвращает массив:

array file ( string $filename

[, int $flags = 0 [, resource $context ]] )

Что за массив возвращает эта функция? Каждый элемент данного массива является строкой в файле, информацию из которого мы считываем (его имя задано аргументом имя\_файла ). *Символ новой строки* тоже включается в каждый из элементов массива. В случае ошибки функция *file()* , как и все уже рассмотренные, возвращает false. Дополнительный аргумент use\_include\_path опять же определяет, искать или нет данный файл в директориях *include\_path*. Открывать удаленные файлы с помощью этой функции тоже можно, если не запрещено сервером. Начиная с PHP 4.3 работа с *бинарными файлами* посредством этой функции стала безопасной.

Например, у нас имеется файл my\_file.html следующего содержания:

<h1>Без труда не вынешь

и рыбку из пруда.</h1>

<b>Тише едешь - дальше будешь</b>

Прочитаем его содержимое с помощью функции file():

<?php

$arr = file ("my\_file.html");

foreach($arr as $i => $a) echo $i,": ",

htmlspecialchars($a), "<br>";

?>

В результате на экран будет выведено следующее сообщение:

0: <h1>Без труда не вынешь

и рыбку из пруда.</h1>

1: <b>Тише едешь - дальше будешь</b>

#### Функция file\_get\_contents

В версиях PHP начиная с 4.3 появилась возможность считывать содержимое файла в строку. Делается это с помощью функции *file\_get\_contents*(). Как и две предыдущие функции, в качестве параметров она принимает значение имени файла и, возможно, указание искать его в директориях *include\_path*. Для порядка все равно приведем ее синтаксис:

string file\_get\_contents ( string $filename

[, bool $use\_include\_path = false

[, resource $context [, int $offset = -1 [, int $maxlen ]]]] )

Эта функция абсолютно идентична функции file(), только возвращает она содержимое файла в виде строки. Кроме того, она безопасна для обработки бинарных данных и может считывать информацию из удаленных файлов, если это не запрещено настройками сервера.

### Проверка существования файла

Итак, создавать *файл* мы научились, записывать данные в него - научились, считывать данные из файла - тоже научились. Но вот вопрос: а что если файла, с которым мы пытаемся проделать все эти *операции*, не существует? Или он недоступен для чтения или записи? Очевидно, что в таком случае ни одна из изученных нами функций работать не будет и *PHP* выдаст *сообщение об ошибке*. Чтобы отслеживать такого рода ошибки, можно использовать функции file\_exists(), is\_writable(), is\_readable().

#### Функция file\_exists

Синтаксис:

bool file\_exists (имя файла или директории)

Функция file\_exists() проверяет, существует ли файл или директория, имя которой передано ей в качестве аргумента. Если директория или файл в файловой системе сервера существует, то функция возвращает TRUE, в противном случае - FALSE. Результат работы этой функции кэшируется. Соответственно очистить кэш можно, как уже отмечалось, с помощью функции clearstatcache(). Для нелокальных файлов использовать функцию file\_exists() нельзя.

<?php

$filename = 'c:/users/files/my\_file.html';

if (file\_exists($filename)) {

print "Файл <b>$filename</b> существует";

} else {

print "Файл <b>$filename</b>

НЕ существует";

}

?>

Пример 9.8. Использование функции file\_exists()

#### Функция is\_writable

Если кроме проверки *существования файла* нужно узнать еще, разрешено ли записывать информацию в этот файл, следует использовать функцию is\_writable() или ее псевдоним - функцию is\_writeable().

Синтаксис:

bool is\_writable (имя файла или директории)

Эта функция возвращает TRUE, если файл (или директория) существует и *доступен для записи*. Доступ к файлу осуществляется под той учетной записью пользователя, под которой работает сервер (чаще всего это пользователь nobody или www). Результаты работы функции is\_writable кэшируются.

#### Функция is\_readable

Если кроме проверки *существования файла* нужно узнать еще, разрешено ли читать информацию из него, нужно использовать функцию is\_readable().

Синтаксис:

bool is\_readable (имя файла)

Эта функция работает подобно функции is\_writable().

<?php

$filename = 'c:/users/files/my\_file.html';

if (is\_readable($filename)) {

print "Файл <b>$filename</b> существует

и доступен для чтения";

} else {

print "Файл <b>$filename</b>

НЕ существует или

НЕ доступен для чтения";

}

?>

Пример 9.9. Использование функции is\_readable()

### Удаление файла

Последнее, что мы хотим изучить из действий над файлами, - это *удаление файлов*. Для того чтобы удалить *файл* с помощью языка *PHP*, нужно воспользоваться функцией unlink(). *Синтаксис* этой функции можно описать следующим образом:

bool unlink ( имя\_файла)

Данная *функция* удаляет *файл*, имеющий имя имя\_файла, возвращает TRUE в случае успеха этой *операции* и FALSE - в случае ошибки. Чтобы удалить *файл*, нужно тоже иметь соответствующие *права* доступа к нему (например, *доступа только на чтение* для *удаления файла*недостаточно).

<?php

$filename = 'c:/users/files/my\_file.html';

unlink($filename);

// удаляем файл с именем

// c:/users/files/my\_file.html

?>

Пример 9.10. Использование функции unlink()

### Загрузка файла на сервер

Теперь решим более сложную и часто возникающую на практике задачу *загрузки файла на сервер*. Первое, что нужно сделать, чтобы загрузить *файл* на *сервер*, это создать *html-форму*. Для того чтобы с помощью этой формы можно было загружать файлы, она должна содержать *атрибут* enctype в теге form со значением multipart/form-data, а также элемент input типа file.

**Пример 9.11**.

<form enctype="multipart/form-data"

action="parse.php" method="post">

<input type="hidden" name="MAX\_FILE\_SIZE"

value="30000" />

Загрузить файл: <input type="file"

name="myfile" /><br>

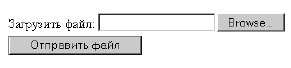
<input type="submit"

value="Отправить файл" />

</form>

Пример 9.11. Форма для загрузки файла на сервер

Заметим, что мы добавили в форме *скрытое поле*, которое содержит в себе максимальный допустимый *размер* загружаемого файла в байтах. При попытке загрузить *файл*, *размер* которого больше указанного в этом *поле* значения, будет зафиксирована ошибка. В браузере созданная нами форма будет выглядеть как строка для ввода текста с дополнительной кнопкой для выбора файла с локального диска ([рис 9.1](http://www.intuit.ru/studies/courses/42/42/lecture/27191?page=6#image.9.1)).



**Рис. 9.1.** Пример формы для загрузки файла на сервер

Теперь нужно написать *скрипт*, который будет обрабатывать полученный *файл*.

Вся *информация* о загруженном на *сервер* файле содержится в глобальном массиве $\_FILES. Этот *массив* появился начиная с *PHP* 4.1.0. Если включена *директива* *register\_globals*, то значения переданных переменных доступны просто *по* их именам.

Если мы загрузили с компьютера-клиента *файл* с именем *critics*.htm *размером* 15136 *байт*, то *скрипт* с единственной командой print\_r($\_FILES) ; выведет на экран следующее:

Array ( [myfile] =>

Array ( [name] => critics.htm

[type] => text/html

[tmp\_name] => C:\WINDOWS\TEMP\php49F.tmp

[error] => 0

[size] => 15136

)

)

Вообще говоря, *массив* $\_FILES всегда имеет следующие элементы:

* $\_FILES['myfile']['name'] - имя, которое имел файл на машине клиента.
* $\_FILES['myfile']['type'] - mime-тип отправленного файла, если браузер предоставил эту информацию. В нашем примере это text/html.
* $\_FILES['myfile']['size'] - *размер* загруженного файла в байтах.
* $\_FILES['myfile']['tmp\_name'] - временное имя файла, под которым он был сохранен на сервере.
* $\_FILES['myfile']['error'] - *код ошибки*, появившейся при загрузке.

Здесь 'myfile' - это имя элемента формы, с помощью которого была произведена *загрузка файла на сервер*. То есть оно может быть другим, если элемент формы назвать иначе. Но вот другие ключи ( name, type и т. д.) остаются неизменными для любой формы.

Если *register\_globals*=On, то доступны также дополнительные переменные, такие как $myfile\_name, которая эквивалентна $\_FILES['myfile']['name'], и т.п.

*Ошибок при загрузке* в *PHP* выделяют пять типов и соответственно $\_FILES['myfile']['error'] может иметь пять значений:

0 - ошибки не произошло, *файл* загружен успешно

1 - загружаемый *файл* превышает *размер*, установленный директивой upload\_max\_filesize в *файле настроек* *php.ini*

2 - загружаемый *файл* превышает *размер*, установленный элементом MAX\_FILE\_SIZE формы html

3 - *файл* был загружен частично

4 - *файл* загружен не был

*По* умолчанию загруженные файлы сохраняются во временной директории сервера, если другая *директория* не указана с помощью опции upload\_tmp\_dir в *файле настроек* *php.ini*. Переместить загруженный *файл* в нужную директорию можно с помощью функции move\_uploaded\_file().

*Функция* move\_uploaded\_file() имеет следующий *синтаксис*:

bool move\_uploaded\_file (временное\_имя\_файла,

место\_назначения )

Эта *функция* проверяет, действительно ли *файл*, обозначенный строкой временное\_имя\_файла, был загружен через механизм загрузки *HTTP* методом *POST*. Если это так, то *файл* перемещается в *файл*, заданный параметром место\_назначения (этот *параметр* содержит как *путь* к новой директории для хранения, так и новое *имя файла*).

Если временное\_имя\_файла задает неправильный загруженный *файл*, то никаких действий произведено не будет, и move\_uploaded\_file() вернет FALSE. То же самое произойдет, если *файл* *по* каким-то причинам не может быть перемещен. В этом случае *интерпретатор* выведет соответствующее предупреждение. Если *файл*, заданный параметром место\_назначения, существует, то *функция* move\_uploaded\_file() перезапишет его.

<?

/\* В версиях PHP, более ранних,

чем 4.1.0, вместо массива

$\_FILES нужно использовать

массив $HTTP\_POST\_FILES \*/

$uploaddir = 'c:/uploads/';

// будем сохранять загружаемые

// файлы в эту директорию

$destination = $uploaddir .

$\_FILES['myfile']['name'];

// имя файла оставим неизменным

print "<pre>";

if (move\_uploaded\_file(

$\_FILES['myfile']['tmp\_name'],

$destination)) {

/\* перемещаем файл из временной папки

в выбранную директорию для хранения \*/

print "Файл успешно загружен <br>";

} else {

echo "Произошла ошибка при загрузке файла.

Некоторая отладочная информация:<br>";

print\_r($\_FILES);

}

print "</pre>";

?>

Пример 9.12. Программа загрузки файла на сервер

### Заключение

Подведем итоги. В этой лекции мы изучили, как создавать файлы с помощью языка *PHP*, как *записывать данные в файлы* посредством *PHP*, как считывать из них информацию различными способами, как проверять *существование* и *доступность файла для записи* и чтения. Кроме того, мы рассмотрели задачу *загрузки файла на сервер* и обсудили основные связанные с ней переменные и функции языка *PHP*.

СЕССИИ

В этой лекции мы разберем, что такое *сессии* и в чем их специфика в *PHP*, решим одну из основных задач, возникающих при построении более-менее сложных информационных систем (сайтов) - задачу *авторизации доступа* пользователей к ресурсам системы, а также обсудим *безопасность* построенного решения.

### Авторизация доступа

Что такое *авторизация доступа*? Попробуем объяснить на примере из обычной жизни. Вы хотите взять в библиотеке книгу. Но эта услуга доступна только тем, у кого есть читательский билет. Можно сказать, что с помощью этого билета производится *"авторизация доступа"* к библиотечным ресурсам. Библиотекарь после предъявления ему читательского билета знает, кто берет книгу, и в случае необходимости (например, книгу долго не возвращают) может принять меры (позвонить должнику домой). Библиотекарь имеет гораздо больше прав, чем обычный посетитель: он может давать или не давать книги определенному посетителю, может выставлять напоказ новинки и убирать в *архив* редко читаемые книги и т.п.

В информационных технологиях все примерно так же. В сети существует огромное количество ресурсов, т.е. множество "библиотек". У каждой из них свой "библиотекарь", т.е. человек или *группа* людей, отвечающих за содержание ресурса и предоставление пользователям информации. Их называют администраторами. Функции администратора, как правило, включают добавление новой информации, удаление и редактирование существующей, настройка способов отображения информации пользователю. А в *функции пользователя* (простого посетителя ресурса) входит только *поиск* и просмотр информации.

Как же отличить пользователя от администратора? В реальной библиотеке это как-то очевидно, но если роли библиотекаря и посетителя библиотеки перенести в виртуальную реальность, то эта очевидность исчезает. Библиотекарь, как и посетитель, имеет *доступ* к библиотечным ресурсам через *Internet*. А согласно протоколу *HTTP* все клиенты абсолютно равноправны. Как же понять, кто зашел на *сайт*? Обычный *пользователь* (посетитель) или *администратор* (библиотекарь)? Если это простой *пользователь*, то как сохранить это *знание*, чтобы не допустить посетителя в закрытые архивы сайта? То есть возникает вопрос, как идентифицировать клиента, который послал *запрос*, и сохранять сведения о нем, пока он находится на сайте?

Самый простой вариант, который приходит в голову, - это *регистрация* человека в системе и выдача ему аналога читательского билета, а именно логина и пароля для входа в административную часть системы. Эта *информация* хранится на компьютере-сервере, и при входе в систему проверяется соответствие введенных пользователем логина и пароля тем, что хранятся в системе. Правда, здесь *по* сравнению с реальной библиотекой ситуация изменяется: читательский билет требуется библиотекарю для входа в закрытую часть системы, а читатель может заходить на *сайт* свободно. В принципе можно регистрировать и простых посетителей. Тогда всех зарегистрированных пользователей нужно разделить на группы: библиотекари (администраторы) и читатели (простые пользователи), наделив их соответствующими правами. Мы не будем вдаваться в эти тонкости и воспользуемся самым простым вариантом, когда ввод логина и пароля требуется для доступа к некоторым страницам сайта.

**Пример 12.1**.

У нас имеется *файл* index.html - домашняя страничка Васи Петрова

<html>

<head><title>My home page</title></head>

<body>

Привет всем!

Меня зовут Вася Петров и

это моя домашняя страничка.

<a href="secret\_info.html">Для Пети</a>

</body></html>

Листинг 12.1. Домашняя страничка Васи Петрова

и *файл* secret\_info.html, который содержит секретную информацию, читать которую разрешено только Васиному другу Пете.

<html>

<head><title>Secret info</title></head>

<body>

Здесь я хочу делиться секретами

с другом Петей.</p>

</body></html>

Листинг 12.1a. secret\_info.html

Если оставить оба эти файла как есть, то любой посетитель, кликнув на ссылку "Для Пети", попадет на секретную страничку. Чтобы этого избежать, нужно добавить промежуточный *скрипт*, который будет проверять, действительно ли Петя хочет попасть на секретную страничку. И сделать так, чтобы главный *файл* ссылался не сразу на secret\_info.html, а сначала на этот *скрипт*.

<html>

<head><title>My home page</title></head>

<body>

<p>Привет всем!

Меня зовут Вася Петров и

это моя домашняя страничка.

</p>

<a href="authorize.php">Для Пети</a>

</body>

</html>

Листинг 12.1b. Index.html

Сам *скрипт* авторизации должен предоставлять форму для ввода логина и пароля, проверять их правильность и перенаправлять на секретную страничку, если проверка прошла успешно, и выдавать *сообщение об ошибке* в противном случае.

<?

if (!isset($\_GET['go'])){

// проверяем, отправлены ли данные формой

echo "<form>

// форма для авторизации

//(ввода логина и пароля)

Login: <input type=text name=login>

Password: <input type=password

name=passwd>

<input type=submit name=go value=Go>

</form>";

}else {

// если форма заполнена, то сравниваем логин

// и пароль с правильными логином и паролем

if ($\_GET['login']=="pit" &&

$\_GET['passwd']=="123") {

Header("Location: secret\_info.html");

//и перенаправляем на секретную страницу

}else echo "Неверный ввод,

попробуйте еще раз<br>";

}

?>

Листинг 12.1c. authorize.php

Вроде бы все достаточно просто. Но допустим, у нас не одна секретная страничка, а несколько. Причем они связаны между собой перекрестными ссылками. Тогда возникает необходимость постоянно помнить *пароль* и логин посетителя сайта (если он таковой имеет). Чтобы решить эту проблему, можно в каждую страницу встроить *скрипт*, который будет передавать логин и *пароль* от страницы к странице в качестве скрытых параметров формы. Но такой способ не совсем безопасен: эти параметры можно перехватить и подделать. В *PHP* существует более удобный и *безопасный метод* решения проблемы хранения данных о посетителе в течение сеанса его работы с сайтом - это механизм *сессий*.

### Механизм сессий

Cессии - это механизм, который позволяет создавать и использовать переменные, сохраняющие свое *значение* в течение всего времени работы пользователя с сайтом.

Эти переменные для каждого пользователя имеют различные значения и могут использоваться на любой странице сайта до выхода пользователя из системы. При этом каждый раз, заходя на *сайт*, *пользователь* получает новые значения переменных, позволяющие идентифицировать его в течение этого сеанса или *сессии* работы с сайтом. Отсюда и название механизма - *сессии*.

Задача *идентификации пользователя* решается путем присвоения каждому пользователю уникального номера, так называемого *идентификатора сессии* (*SID*, *Session* IDentifier). Он генерируется *PHP* в тот момент, когда *пользователь* заходит на *сайт*, и уничтожается, когда *пользователь* уходит с сайта, и представляет собой строку из 32 символов (например, ac4f4a45bdc893434c95dcaffb1c1811). Этот *идентификатор* передается на *сервер* вместе с каждым запросом клиента и возвращается обратно вместе с ответом сервера.

Существует несколько способов передачи *идентификатора сессии*:

* С помощью *cookies*.

*Cookies* были созданы специально как метод однозначной идентификации клиентов и представляют собой расширение протокола HTTP. В этом случае *идентификатор сессии* сохраняется во временном файле на компьютере клиента, пославшего запрос. Метод, несомненно, хорош, но многие пользователи отключают поддержку *cookies* на своем компьютере из-за проблем с безопасностью.

* С помощью параметров адресной строки.

В этом случае *идентификатор сессии* автоматически встраивается во все запросы (URL), передаваемые серверу, и хранится на стороне сервера.

Например: *адрес* <http://green.nsu.ru/test.php> превращается в *адрес* <http://green.nsu.ru/test.php?PHPSESSID=ac4f4a45bdc893434c95dcaffb1c1811>

Этот способ передачи идентификатора используется автоматически, если у браузера, отправившего *запрос*, выключены *cookies*. Он достаточно надежный - передавать параметры в адресной строке можно всегда. С другой стороны, *идентификатор сессии* можно подглядеть, воспользоваться сохраненным вариантом в строке браузера или подделать. Хотя, конечно, все эти проблемы либо надуманны либо их можно решить. Например, кто сможет запомнить строку из 32 различных символов? А если правильно организовать работу с *сессиями* (вовремя их уничтожать), то даже сохранившийся в браузере номер *сессии* ничего не даст. К вопросам безопасности мы еще вернемся в конце лекции.

Кроме перечисленных вариантов передачи *идентификатора сессии*, известно еще несколько, но мы их рассматривать не будем ввиду их сложности.

### Настройка сессий

Прежде чем начать работать с *сессиями*, следует разобраться в том, как корректно настраивать их обработку интерпретатором *PHP*. Сама работа с *сессиями* в *PHP* поддерживается *по* умолчанию. Это значит, что устанавливать никаких дополнительных элементов не нужно. А вот знать, что записано в настройках этого модуля, полезно, чтобы избежать ошибок при работе с ним.

Настройки *PHP*, в том числе и для работы с *сессиями*, прописываются в файле *php.ini*. Обратимся к этому файлу.

Как мы уже знаем, *идентификатор сессии* (число, *по* которому можно уникально идентифицировать клиента, пославшего *запрос*) сохраняется либо на компьютере-сервере, либо на компьютере-клиенте, либо и там, и там.

Параметр ***session.save\_path*** в *php.ini*, определяет, где на сервере будут храниться данные *сессии* . Из-за него чаще всего возникают проблемы для Windows-серверов, потому что *по* умолчанию *значение* *session.save\_path* установлено в /tmp. И если в корневой директории сервера такой папки нет, то при запуске *сессий* будет выдаваться ошибка.

*Сервер* может обрабатывать большое количество *сессий* одновременно, и все их временные файлы будут храниться в директории, заданной параметром *session.save\_path*. Если система плохо работает с папками большого размера, то удобно использовать поддиректории. Для этого, кроме названия папки, в *значение* параметра добавляют еще и число, определяющее глубину вложенности поддиректорий в этой папке: N;/dir. Это *значение* нужно обязательно взять в кавычки, поскольку точка с запятой является одним из символов комментариев в *файле настроек* *PHP*. Все директории и поддиректории для хранения данных *сессии* нужно создать самостоятельно.

Например: 2;/Temp определяет, что переменные *сессий* будут храниться в папках вида c:\Temp\0\a\, c:\Temp\0\b\ и т.п.

*Хранение данных* на стороне клиента осуществляется с помощью *cookies*. Работу *PHP* с *cookies* можно настроить, в частности, с помощью параметров *session.use\_cookies*, *session.cookie\_lifetime* и т.п.

Параметр ***session.use\_cookies*** определяет, использовать ли *cookies* при работе с *сессиями*. По умолчанию эта опция включена (т.е. принимает значение "1" ).

Параметр ***session.cookie\_lifetime*** задает длительность жизни *cookies* в секундах. По умолчанию это "0", т.е. данные в *cookies* считаются правильными до закрытия окна браузера.

Кроме этих параметров, полезными могут оказаться *session.name*, определяющий имя *сессии*, ***session.auto\_start***, позволяющий автоматически запускать *сессии*, ***session.serialize\_handler***, задающий способ кодировки данных *сессии*, и *параметр* ***session.cache\_expire***, определяющий, через сколько минут устаревает документ в кэше.

Имя *сессии* *session.name* *по* умолчанию устанавливается как PHPSESSID и используется в *cookies* как *имя переменной*, в которой хранится *идентификатор сессии*. Автоматический *запуск* *сессий* *по* умолчанию отключен, но его можно задать, сделав *значение* *session.auto\_start*равным "1". Для кодирования данных *сессии* *по* умолчанию используется *php*. Устаревание данных, сохраненных в кэше, происходит через 180 минут.

Существует еще множество настроек, с которыми можно познакомиться в документации или непосредственно в *файле настроек* *php.ini*. На наш взгляд, знакомства с перечисленными выше параметрами достаточно для работы с *сессиями* в *PHP*. Так что приступим.

### Работа с сессиями

#### Создание сессии

Первое, что нужно сделать для работы с *сессиями* (если они уже настроены администратором сервера), это запустить механизм *сессий*. Если в настройках сервера переменная *session.auto\_start* установлена в значение "0" (если *session.auto\_start*=1, то *сессии* запускаются автоматически), то любой скрипт, в котором нужно использовать данные *сессии*, должен начинаться с команды

session\_start();

Получив такую команду, сервер создает новую *сессию* или восстанавливает текущую, основываясь на *идентификаторе сессии*, переданном по запросу. Как это делается? Интерпретатор PHP ищет переменную, в которой хранится *идентификатор сессии* (по умолчанию это PHPSESSID) сначала в *cookies*, потом в переменных, переданных с помощью POST- и GET-запросов. Если идентификатор найден, то пользователь считается идентифицированным, производится замена всех URL и выставление *cookies*. В противном случае пользователь считается новым, для него генерируется новый уникальный идентификатор, затем производится замена URL и выставление *cookies*.

Команду session\_start() нужно вызывать во всех скриптах, в которых предстоит использовать переменные *сессии*, причем до вывода каких-либо данных в браузер. Это связано с тем, что *cookies* выставляются только до вывода информации на экран.

Получить идентификатор текущей *сессии* можно с помощью *функции session\_id()*.

Для наглядности *сессии* можно задать имя с помощью функции *session\_name*([имя\_сессии]). Делать это нужно еще до инициализации *сессии*. Получить имя текущей *сессии* можно с помощью этой же функции, вызванной без параметров: *session\_name();*

**Пример 12.2**. Создание сессии

Переименуем наш файл index.html, чтобы обрабатывались *php-скрипты*, например в Index.php, создадим *сессию* и посмотрим, какой она получит идентификатор и имя.

<?

session\_start();

// создаем новую сессию или

// восстанавливаем текущую

echo session\_id();

// выводим идентификатор сессии

?>

<html>

<head><title>My home page</title></head>

... // домашняя страничка

</html>

<?

echo session\_name();

// выводим имя текущей сессии.

// В данном случае это PHPSESSID

?>

Пример 12.2. Создание сессии

Если проделать то же самое с файлом authorize.php, то значения выводимых переменных (id *сессии* и ее имя) будут такими же, если перейти на него с index.php и не закрывать перед этим окно браузера (тогда *идентификатор сессии* изменится).

#### Регистрация переменных сессии

Однако от самих идентификатора и имени *сессии* нам пользы для решения наших задач немного. Мы же хотим передавать и сохранять в течение *сессии* наши собственные переменные (например, логин и пароль). Для того чтобы этого добиться, нужно просто зарегистрировать свои переменные:

session\_register(имя\_переменной1,

имя\_переменной2, ...);

Заметим, что регистрируются не значения, а имена переменных. Зарегистрировать переменную достаточно один раз на любой странице, где используются *сессии*. Имена переменных передаются *функции session\_register()* без знака $. Все зарегистрированные таким образом переменные становятся глобальными (т.е. доступными с любой страницы) в течение данной *сессии* работы с сайтом.

Зарегистрировать переменную также можно, просто записав ее значение в ассоциативный *массив $\_SESSION*, т.е. написав

$\_SESSION['имя\_переменной'] =

'значение\_переменой';

В этом массиве хранятся все зарегистрированные (т.е. глобальные) переменные *сессии*.

Доступ к таким переменным осуществляется с помощью массива $\_SESSION['имя\_переменной'] (или $HTTP\_SESSION\_VARS['имя\_переменной'] для версии PHP 4.0.6 и более ранних). Если же в настройках php включена опция *register\_globals*, то к сессионным переменным можно обращаться еще и как к обычным переменным, например так: $имя\_переменной.

Если *register\_globals*=off (отключены), то пользоваться *session\_register()* для регистрации переменных, переданных методами POST или GET, нельзя, т.е. это просто не работает. И вообще, не рекомендуется одновременно использовать оба метода *регистрации переменных*, *$\_SESSION* и *session\_register()*. (Начиная с версии PHP 5.3.0 не рекомендуется для *регистрации переменных сессии* использовать функцию session\_register() ; более того, начиная с версии PHP 6.0.0, эта функция станет недоступна. Вместо этого, для *регистрации переменных сессии* рекомендуется пользоваться массивом $\_SESSION.)

**Пример 12.3**. Регистрация переменных

Зарегистрируем логин и пароль, вводимые пользователем на странице авторизации.

<?

session\_start();

// создаем новую сессию или

// восстанавливаем текущую

if (!isset($\_GET['go'])){

echo "<form>

Login: <input type=text name=login>

Password: <input type=password

name=passwd>

<input type=submit name=go value=Go>

</form>";

}else {

$\_SESSION['login']=$\_GET['login'];

// регистрируем переменную login

$\_SESSION['passwd']=$\_GET['passwd'];

// регистрируем переменную passwd

// теперь логин и пароль - глобальные

// переменные для этой сессии

if ($\_GET['login']=="pit" &&

$\_GET['passwd']=="123") {

Header("Location: secret\_info.php");

// перенаправляем на страницу

// secret\_info.php

}else echo "Неверный ввод,

попробуйте еще раз<br>";

}

print\_r($\_SESSION);

// выводим все переменные сессии

?>

Листинг 12.3a. authorize.php

Теперь, попав на страничку secret\_info.php, да и на любую другую страницу сайта, мы сможем работать с введенными пользователем логином и паролем, которые будут храниться в *массиве $\_SESSION*. Таким образом, если изменить код секретной странички (заметьте, мы переименовали ее в secret\_info.php) так:

<?php

session\_start();

// создаем новую сессию или

// восстанавливаем текущую

print\_r($\_SESSION);

// выводим все переменные сессии

?>

<html>

<head><title>Secret info</title></head>

<body>

<p>Здесь я хочу делиться секретами

с другом Петей.

</body>

</html>

Листинг 12.3b. secret\_info.php

То мы получим в браузере на секретной странице следующее:

Array ( [login] => pit [passwd] => 123 )

Здесь я хочу делиться секретами

с другом Петей.

В итоге получим *список переменных*, зарегистрированных на authorize.php и, собственно, саму секретную страничку.

Что это нам дает? Допустим, хакер хочет прочитать секреты Васи и Пети. И он как-то узнал, как называется секретная страничка (или странички). Тогда он может попытаться просто ввести ее адрес в строке браузера, минуя страницу авторизации (ввода пароля). Чтобы избежать такого проникновения в наши тайны, нужно дописать всего пару строк в код секретных страничек:

<?php

session\_start();

// создаем новую сессию или

// восстанавливаем текущую

print\_r($\_SESSION);

// выводим все переменные сессии

if (!($\_SESSION['login']=="pit" &&

$\_SESSION['passwd']==123))

// проверяем правильность

// пароля-логина

Header("Location: authorize.php");

// если ошибка, то перенаправляем на

// страницу авторизации

?>

<html>

<head><title>Secret info</title></head>

... // здесь располагается

//секретная информация :)

</html>

Листинг 12.3c. 2-я версия secret\_info.php

#### Удаление переменных сессии

Кроме умения регистрировать переменные *сессии* (т.е. делать их глобальными на протяжении всего сеанса работы), полезно также уметь удалять такие переменные и *сессию* в целом.

***Функция session\_unregister(имя\_переменной)*** удаляет глобальную переменную из текущей *сессии* (т.е. удаляет ее из списка зарегистрированных переменных). Если регистрация производилась с помощью *$\_SESSION* ( *$HTTP\_SESSION\_VARS* для версии PHP 4.0.6 и более ранних), то используют языковую конструкцию unset(). Она не возвращает никакого значения, а просто уничтожает указанные переменные.

Где это может пригодиться? Например, для уничтожения данных о посетителе (в частности, логина и пароля) после его ухода с секретной странички. Если правильные логин и пароль сохранятся и окно браузера после посещения сайта не закрыли, то любой другой пользователь этого компьютера сможет прочитать закрытую информацию.

**Пример 12.4**. Уничтожение переменных сессии

В файл secret\_info.php добавим строчку для выхода на главную страницу:

<?php

// ... php код

?>

<html>

<head><title>Secret info</title></head>

... // здесь располагается

// секретная информация :)

<a href="index.php">На главную</a>

</html>

Пример 12.4a. secret\_info.php

В Index.php уничтожим логин и пароль, введенные ранее:

<?

session\_start();

session\_unregister('passwd');

// уничтожаем пароль

unset($\_SESSION['login']);

// уничтожаем логин

print\_r($\_SESSION);

// выводим глобальные переменные сессии

?>

<html>

<head><title>My home page</title></head>

... // домашняя страничка

</html>

Пример 12.4b. Index.php

Теперь, чтобы попасть на секретную страницу, нужно будет опять вводить логин и пароль.

Для того чтобы сбросить значения всех переменных *сессии*, можно использовать *функцию session\_unset();*

Уничтожить текущую *сессию* целиком можно командой *session\_destroy();* Она не сбрасывает значения глобальных переменных *сессии* и не удаляет *cookies*, а уничтожает все данные, ассоциируемые с текущей *сессией*.

<?

session\_start(); // инициализируем сессию

$test = "Переменная сессии";

$\_SESSION['test']= $test;

// регистрируем переменную $test.

// если register\_globals=on,

// то можно использовать

// session\_register('test');

print\_r($\_SESSION);

// выводим все глобальные переменные

echo session\_id();

// выводим идентификатор сессии

echo "<hr>";

session\_unset();

// уничтожаем все глобальные

// переменные сессии

print\_r($\_SESSION);

echo session\_id();

echo "<hr>";

session\_destroy(); // уничтожаем сессию

print\_r($\_SESSION);

echo session\_id();

?>

Пример 12.5. Уничтожение сессии и глобальных переменных

В результате работы этого скрипта будут выведены три строки: в первой - массив с элементом test и его значением, а также *идентификатор сессии*, во второй - пустой массив и *идентификатор сессии*, в третьей - пустой массив. Таким образом, видно, что после уничтожения *сессии* уничтожается и ее идентификатор, и мы больше не можем ни регистрировать переменные, ни вообще производить какие-либо действия с *сессией*.

### Безопасность

Вообще говоря, cледует понимать, что использование механизма *сессий* не гарантирует полной *безопасности системы*. Для этого нужно принимать дополнительные меры. Обратим внимание на проблемы с безопасностью, которые могут возникнуть при работе с *сессиями* и, в частности, с теми программами, что мы написали.

Во-первых, опасно передавать туда-сюда *пароль*, его могут перехватить. Кроме того, мы зарегистрировали его как глобальную переменную *сессии*, значит, он сохранился в *cookies* на компьютере-клиенте. Это тоже плохо. И вообще, пароли и логины *по*-хорошему должны храниться в базе данных. Пусть *информация* о пользователях хранится в базе данных "test" (в таблице "users"), а мы имеем к ней *доступ* под логином my\_user и паролем my\_passwd.

Во-вторых, что делать, если кто-то написал *скрипт* подбора пароля для секретной страницы? В этом случае на страницу авторизации много раз должен стучаться какой-то посторонний *скрипт*. Поэтому нужно просто проверять, с нашего ли сайта пришел *запрос* на авторизацию, и если нет, то не пускать его дальше. *Адрес* страницы, с которой поступил *запрос*, можно получить с помощью глобальной переменной *$\_SERVER['HTTP\_REFERER'])*. Хотя, конечно, если за взлом сайта взялись всерьез, то *значение* этой переменной тоже подменят (например, с помощью того же *PHP*). Тем не менее проверку ее значения можно считать одним из важнейших шагов на пути к обеспечению безопасности своего сайта.

<?

session\_start();

// создаем новую сессию или

// восстанавливаем текущую

$conn = mysql\_connect("localhost",

"my\_user","my\_passwd");

// устанавливаем соединение с сервером БД

mysql\_select\_db("test");

// выбираем рабочую базу данных

$SERVER\_ROOT = "http://localhost/~nina/tasks/sessions/";

// где находятся наши скрипты

/\* с помощью регулярного выражения

^$SERVER\_ROOT и функции eregi проверяем,

начинается ли адрес ссылающегося скрипта,

т.е. строка $\_SERVER['HTTP\_REFERER'])

со строки $SERVER\_ROOT (как у нас) \*/

if(eregi("^$SERVER\_ROOT",

$\_SERVER['HTTP\_REFERER'])){

// если да, то делаем почти то же, что и

// раньше, пароль регистрировать не будем

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

echo "<form method=POST >

Login: <input type=text name=login>

Password: <input type=password name=passwd>

<input type=submit name=go value=Go>

</form>";

}else {

/\* запрос к базе данных: выбираем из таблицы

users login, который совпадает с переданным

по запросу, причем пароль у него тоже должен

совпасть с введенным пользователем.

Если этого нет, то считаем, что логин и

пароль введены неверно \*/

$sql = "SELECT login FROM users

WHERE login='" . $\_POST['login'] . "' AND passwd='" . $\_POST['passwd'] . "';";

$q = mysql\_query($sql,$conn); // отправляем запрос к БД

$n = mysql\_num\_rows($q); // число строк в ответе на запрос

if (!$n==0){

$\_SESSION['user\_login']=$\_POST['login'];

// регистрируем переменную login

Header("Location: secret\_info.php");

// перенаправляем на страницу secret\_info.php

}else echo "Неверный ввод, попробуйте еще раз<br>";

}

print\_r($\_SESSION); // выводим все переменные сессии

}

?>

Листинг 12.6. authorize.php

Вроде бы первые две проблемы решены. Но есть еще одна. Что делать, если *хакер* просто допишет в строку запроса *значение* какой-нибудь глобальной переменной (например, логина)? Вообще это возможно, только если *register\_globals*=On. Просто иначе мы используем для работы с глобальными переменными *массив $\_SESSION* и с ним такие фокусы не проходят. Все же попробуем решить и эту проблему. Для этого нужно очистить строку запроса перед тем, как сравнивать значения параметров. То есть сначала сбросим *значение* $user\_login. Потом данную переменную нужно опять зарегистрировать, но не как новую, а как уже существующую. Для этого знак доллара при регистрации НЕ опускается. Вот что получилось:

<?php

unset($user\_login); // уничтожаем переменную

session\_start(); // создаем новую сессию или

// восстанавливаем текущую

session\_register($user\_login);

// регистрируем переменную

// как уже существующую

if (!($user\_login=="pit")) // проверяем логин

Header("Location: authorize.php");

// если ошибка, то перенаправляем

// на страницу авторизации

?>

<html>

<head><title>Secret info</title></head>

... // здесь располагается

// секретная информация :)

</html>

Листинг 12.7. secret\_info.php

### Заключение

Итак, мы познакомились с *сессиями* и основными способами работы с ними, проблемами, возникающими при их использовании, и возможными решениями этих проблем. Надеюсь, что после прочтения лекции читателям стало ясно, насколько удобны и просты в использовании *сессии*, а приведенные примеры пригодятся на практике.

**4.5 Работа с базами данных MySQL**

### Базы данных: основные понятия

В жизни мы часто сталкиваемся с необходимостью хранить какую-либо информацию, а потому часто имеем дело и с *базами данных*. Например, мы используем записную книжку для хранения номеров телефонов своих друзей и планирования своего времени. Телефонная книга содержит информацию о людях, живущих в одном городе. Все это своего рода *базы данных*. Ну а раз это *базы данных*, то посмотрим, как в них хранятся данные. Например, телефонная книга представляет собой таблицу ([табл. 10.1](http://www.intuit.ru/studies/courses/42/42/lecture/27193?page=1#table.10.1)).

В этой таблице данные – это собственно номера телефонов, адреса и ФИО., т.е. строки "Иванов Иван Иванович", "32-43-12" и т.п., а названия столбцов этой таблицы, т.е. строки "ФИО", "Номер телефона" и "*Адрес*" задают смысл этих данных, их семантику.

|  |  |  |
| --- | --- | --- |
| Таблица 10.1. Пример базы данных: телефонная книга | | |
| **ФИО** | **Номер телефона** | **Адрес** |
| Иванов Иван Иванович | 32-43-12 | ул. Ленина, 12, 43 |
| Ильин Федор Иванович | 32-32-34 | пр. Маркса, 32, 45 |

Теперь представьте, что записей в этой таблице не две, а две тысячи, вы занимаетесь созданием этого справочника и где-то произошла ошибка (например, опечатка в адресе). Видимо, тяжеловато будет найти и исправить эту ошибку вручную. Нужно воспользоваться какими-то *средствами автоматизации*. Для управления большим количеством данных программисты (не без помощи математиков) придумали системы управления *базами данных* ( *СУБД* ). *По* сравнению с текстовыми *базами данных* электронные *СУБД* имеют огромное число преимуществ, от возможности быстрого поиска информации, взаимосвязи данных между собой до использования этих данных в различных прикладных программах и одновременного доступа к данным нескольких пользователей.

Для точности дадим *определение* *базы данных*, предлагаемое *Глоссарий*.ру

**База данных** – это совокупность связанных данных, организованных по определенным правилам, предусматривающим общие принципы описания, хранения и манипулирования, независимая от прикладных программ. *База данных* является *информационной моделью предметной области*. Обращение к *базам данных* осуществляется с помощью системы управления *базами данных* ( **СУБД** ). *СУБД*обеспечивает поддержку создания *баз данных*, *централизованного управления* и организации доступа к ним различных пользователей.

Итак, мы пришли к выводу, что хранить данные независимо от программ, так, что они связаны между собой и организованы *по*определенным правилам, целесообразно. Но вопрос, как хранить данные, *по* каким правилам они должны быть организованы, остался открытым. Способов существует множество (кстати, называются они моделями представления или хранения данных). Наиболее популярные – объектная и *реляционная* модели данных.

Автором **реляционной модели** считается Э. Кодд, который первым предложил использовать для обработки данных аппарат теории множеств (*объединение*, *пересечение*, *разность*, *декартово произведение*) и показал, что любое *представление* данных сводится к совокупности двумерных таблиц особого вида, известного в математике как *отношение*.

Таким образом, реляционная *база данных* представляет собой набор таблиц (точно таких же, как приведенная выше), связанных между собой. Строка в таблице соответствует сущности реального мира (в приведенном выше примере это *информация* о человеке).

Примеры реляционных *СУБД*: *MySql*, PostgreSql.

В основу **объектной модели** положена концепция объектно-ориентированного программирования, в которой данные представляются в виде набора объектов и классов, связанных между собой родственными отношениями, а работа с объектами осуществляется с помощью скрытых (инкапсулированных) в них методов.

Примеры объектных *СУБД*: *Cache*, GemStone (от Servio Corporation), ONTOS (ONTOS).

В последнее время производители *СУБД* стремятся соединить два этих подхода и проповедуют объектно-реляционную модель представления данных. Примеры таких *СУБД* – *IBM* *DB2* for Common Servers, Oracle8.

### СУБД MySQL

Продолжим разговор о *СУБД* *MySQL*. ***MySQL*** – это реляционная система управления *базами данных*. То есть данные в ее базах хранятся в виде логически связанных между собой таблиц, доступ к которым осуществляется с помощью языка запросов *SQL* . *MySQL* – свободно распространяемая система, т.е. платить за ее применение не нужно. Кроме того, это достаточно быстрая, надежная и, главное, простая в использовании *СУБД*, вполне подходящая для не слишком глобальных проектов.

Работать с *MySQL* можно не только в текстовом режиме, но и в графическом. Существует очень популярный визуальный интерфейс (кстати, написанный на PHP) для работы с этой *СУБД*. Называется он ***PhpMyAdmin***. Этот интерфейс позволяет значительно упростить работу с *базами данных* в *MySQL* .

В текстовом режиме работа с *базой данных* выглядит просто как ввод команд в командную строку ([рис 10.2](http://www.intuit.ru/studies/courses/42/42/lecture/27193?page=2#image.10.2)), а результаты выборок возвращаются в виде своеобразных таблиц, поля в которых налезают друг на друга, если данные не помещаются на экран ([рис 10.3](http://www.intuit.ru/studies/courses/42/42/lecture/27193?page=2#image.10.3)).

Работа с MySQL в командной строке. Команда show databases — вывести все имеющиеся базы данных

**Рис. 10.2.** Работа с MySQL в командной строке. Команда show databases — вывести все имеющиеся базы данных

*PhpMyAdmin* позволяет пользоваться всеми достоинствами браузера, включая прокрутку изображения, если оно не умещается на экран. Многие из базовых *SQL* -функций работы с данными в *PhpMyAdmin* сведены к интуитивно понятным интерфейсам и действиям, напоминающим переход *по* ссылкам в *Internet*. Но, тем не менее, стоит все же поработать и в текстовом режиме.



**Рис. 10.3.** Работа с MySQL в командной строке. Результат обработки команды show databases

Перед тем как переходить к детальному изучению языка *SQL*, несколько слов об установке *MySQL* и подготовке к работе. Если вы не собираетесь заниматься администрированием сервера, то *информация*, приведенная ниже, пригодится вам только для общего развития. Итак, устанавливается *MySQL* очень просто – автоматически, пару раз нажмите OK, и все. После этого вы можете зайти в директорию, где лежат файлы типа *mysql*.exe, mysqld.exe и т.п. (у нас под *Windows* XP это c:\*mysql*\bin ) Последний *файл* запускает *Mysql*-*сервер*. В некоторых системах *сервер* запускается в виде сервиса. После запуска сервера следует запустить *mysql*-клиент, запустив программу *mysql*.exe. Здесь даже пароля не спросят. Более того, если вы наберете

shell> mysql.exe -u root

или

shell>mysql -u root mysql

то получите все *права* администратора *mysql* сервера. Кстати, выполнять эти команды надо, находясь в той директории, где лежат файлы *mysql*.exe.

Для начала, не вдаваясь в подробности команд, исправим эти два недочета (отсутствие пароля у администратора и возможность входа анонимным пользователям):

shell> mysql -u root mysql

mysql> UPDATE user SET Password=PASSWORD('new\_password')

WHERE user='root';

mysql> DELETE FROM user WHERE user='';

mysql> FLUSH PRIVILEGES;

Все данные о пользователях *MySQL* хранит в таблице user в специальной *базе данных* *mysql*, *доступ* к которой имеет только *администратор* сервера. Поэтому, чтобы изменить какой-либо *пароль*, нужно изменить эту таблицу. *Пароль* задается с помощью функции PASSWORD, которая кодирует введенные данные. Кроме изменения пароля администратора, нужно еще удалить всех пользователей, не имеющих логина (*команда* *DELETE* ). *Команда* *Flush* *Privileges* заставляет вступить в действие изменения, произошедшие в системной *базе данных* (*mysql*).

Теперь создадим *базу данных*, с которой будем работать (мы все еще работаем как *администратор* сервера):

mysql>create database book;

Как можно заметить, все команды в *MySQL* заканчиваются точкой с запятой. Если вы забыли поставить этот знак, то выдается приглашение его поставить до тех пор, пока это не будет сделано:

mysql> show tables

->

->

Теперь последнее действие – создадим простого пользователя, предоставим ему *доступ* к созданной *базе данных*, и начнем работать.

mysql> GRANT ALL PRIVILEGES ON book.\* TO nina@localhost

IDENTIFIED BY '123';

*Команда* *GRANT* наделяет пользователя nina, зашедшего на *сервер* с этой же машины (c *localhost*) и идентифицируемого паролем "123", определенными правами (в данном случае всеми) на все таблицы *базы данных* book. Теперь мы можем выйти и зайти как *пользователь*nina с соответствующим паролем:

shell>mysql -u nina -p

Enter password: \*\*\*

Welcome to the MySQL monitor!...

mysql>

Если вы собираетесь пользоваться *базой данных* на чужом сервере, то его *администратор* проделает все описанные выше действия за вас, т.е. все настроит и создаст пользователя и *базу данных*. В следующей главе описаны команды языка *SQL*, которые пригодятся для работы с данными, хранящимися в *СУБД* *MySQL*.

### Построение интерфейса для добавления информации

Итак, у нас есть какая-то *таблица* в базе данных. Чтобы построить *интерфейс* для добавления информации в эту таблицу, нужно ее структуру (т.е. набор ее полей) отобразить в *html-форму*.

Разобьем эту задачу на следующие подзадачи:

* *установка соединения с БД* ;
* *выбор рабочей БД* ;
* *получение списка полей таблицы* ;
* отображение полей в *html-форму*.

После этого данные, введенные в форму, нужно записать в базу данных. Рассмотрим все эти задачи по порядку.

#### Установка соединения

Итак, первое, что нужно сделать, - это установить соединение с базой данных. Воспользуемся функцией *mysql\_connect*.

Синтаксис *mysql\_connect*

ресурс mysql\_connect ( [строка server

[, строка username [, строка password

[, логическое new\_link

[, целое client\_flags]]]]])

Данная функция устанавливает соединение с сервером *MySQL* и возвращает указатель на это соединение или FALSE в случае неудачи. Для отсутствующих параметров устанавливаются следующие значения по умолчанию:

server = 'localhost:3306'

username = имя пользователя владельца

процесса сервера

password = пустой пароль

Если функция вызывается дважды с одними и теми же параметрами, то новое соединение не устанавливается, а возвращается ссылка на старое соединение. Чтобы этого избежать, используют параметр new\_link, который заставляет в любом случае открыть еще одно соединение.

Параметр client\_flags - это комбинация следующих констант: *MYSQL\_CLIENT\_COMPRESS* (использовать протокол сжатия), *MYSQL\_CLIENT\_IGNORE\_SPACE* (позволяет вставлять пробелы после имен функций), *MYSQL\_CLIENT\_INTERACTIVE* (ждать interactive\_timeout секунд - вместо wait\_timeout - до закрытия соединения).

Параметр new\_link появился в PHP 4.2.0, а параметр client\_flags - в PHP 4.3.0.

Соединение с сервером закрывается при завершении исполнения скрипта, если оно до этого не было закрыто с помощью функции *mysql\_close()*.

Итак, устанавливаем соединение с базой данных на локальном сервере для пользователя nina с паролем "123":

<?

$conn = mysql\_connect(

"localhost", "nina","123")

or die("Невозможно установить

соединение: ". mysql\_error());

echo "Соединение установлено";

mysql\_close($conn);

?>

Действие *mysql\_connect* равносильно команде

shell>mysql -u nina -p123

#### Выбор базы данных

После *установки соединения* нужно выбрать базу данных, с которой будем работать. Наши данные хранятся в базе данных book. В *MySQL выбор базы данных* осуществляется с помощью команды use:

mysql>use book;

В PHP для этого существует функция *mysql\_select\_db*.

Синтаксис *mysql\_select\_db*:

логическое mysql\_select\_db (

строка database\_name

[, ресурс link\_identifier])

Эта функция возвращает TRUE в случае успешного *выбора базы данных* и FALSE - в противном случае.

Сделаем базу данных book рабочей:

<?

$conn = mysql\_connect(

"localhost","nina","123")

or die("Невозможно установить

соединение: ". mysql\_error());

echo "Соединение установлено";

mysql\_select\_db("book");

?>

#### Получение списка полей таблицы

Теперь можно заняться собственно решением задачи. Как получить список полей таблицы? Очень просто. В PHP и на этот случай есть своя команда - mysql\_list\_fields.

Синтаксис mysql\_list\_fields

ресурс mysql\_list\_fields (

строка database\_name,

строка table\_name

[, ресурс link\_identifier])

Эта функция возвращает список полей в таблице table\_name в базе данных database\_name. Получается, что выбирать базу данных нам было необязательно, но это пригодится позже. Как можно заметить, результат работы этой функции - переменная типа ресурс. То есть это не совсем то, что мы хотели получить. Это ссылка, которую можно использовать для получения информации о полях таблицы, включая их названия, типы и флаги.

Функция *mysql\_field\_name* возвращает имя поля, полученного в результате выполнения запроса. Функция *mysql\_field\_len* возвращает длину поля. Функция *mysql\_field\_type* возвращает тип поля, а функция *mysql\_field\_flags* возвращает список флагов поля, записанных через пробел. Типы поля могут быть int, real, string, *blob* и т.д. Флаги могут быть not\_null, primary\_key, unique\_key, *blob*, auto\_increment и т.д.

Синтаксис у всех этих команд одинаков:

строка mysql\_field\_name (

ресурс result, целое field\_offset)

строка mysql\_field\_type (

ресурс result, целое field\_offset)

строка mysql\_field\_flags (

ресурс result, целое field\_offset)

строка mysql\_field\_len (

ресурс result, целое field\_offset)

Здесь result - это идентификатор результата запроса (например, запроса, отправленного функциями mysql\_list\_fields или *mysql\_query* (о ней будет рассказано позднее)), а field\_offset - порядковый номер поля в результате.

Вообще говоря, то, что возвращают функции типа mysql\_list\_fields или *mysql\_query*, представляет собой таблицу, а точнее, указатель на нее. Чтобы получить из этой таблицы конкретные значения, нужно задействовать специальные функции, которые построчно читают эту таблицу. К таким функциям и относятся *mysql\_field\_name* и т.п. Чтобы перебрать все строки в таблице результата выполнения запроса, нужно знать число строк в этой таблице. Команда *mysql\_num\_rows* (ресурс result ) возвращает число строк во множестве результатов result.

А теперь попробуем получить список полей таблицы *Artifacts* (коллекция экспонатов).

<?

$conn = mysql\_connect(

"localhost","nina","123")

or die("Невозможно установить

соединение: ". mysql\_error());

echo "Соединение установлено";

mysql\_select\_db("book");

$list\_f = mysql\_list\_fields (

"book","Artifacts",$conn);

$n = mysql\_num\_fields($list\_f);

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

$type = mysql\_field\_type($list\_f, $i);

$name\_f = mysql\_field\_name($list\_f,$i);

$len = mysql\_field\_len($list\_f, $i);

$flags\_str = mysql\_field\_flags (

$list\_f, $i);

echo "<br>Имя поля: ". $name\_f;

echo "<br>Тип поля: ". $type;

echo "<br>Длина поля: ". $len;

echo "<br>Строка флагов поля: ".

$flags\_str . "<hr>";

}

?>

В результате должно получиться примерно вот что (если в таблице всего два поля, конечно):

Имя поля: id

Тип поля: int

Длина поля: 11

Строка флагов поля:

not\_null primary\_key auto\_increment

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Имя поля: title

Тип поля: string

Длина поля: 255

Строка флагов поля:

#### Отображение списка полей в html-форму

Теперь немножко подкорректируем предыдущий пример. Будем не просто выводить информацию о поле, а отображать его в подходящий элемент *html-формы*. Так, элементы типа *BLOB* переведем в textarea (заметим, что поле description, которое мы создавали с типом TEXT, отображается как имеющее тип *BLOB* ), числа и строки отобразим в текстовые строки ввода <input type=text>, а элемент, имеющий метку автоинкремента, вообще не будем отображать, поскольку его значение устанавливается автоматически.

Все это решается довольно просто, за исключением выделения из списка флагов флага auto\_increment. Для этого нужно воспользоваться функцией *explode*.

Синтаксис *explode*:

массив explode( строка separator,

строка string [, int limit])

Эта функция разбивает строку string на части с помощью разделителя separator и возвращает массив полученных строк.

В нашем случае в качестве разделителя нужно взять пробел " ", а в качестве исходной строки для разбиения - строку флагов поля.

Итак, создадим форму для ввода данных в таблицу *Artifacts*:

<?

$conn=mysql\_connect("localhost","nina","123");

// устанавливаем соединение

$database = "book";

$table\_name = "Artifacts";

mysql\_select\_db($database); // выбираем базу данных для

// работы

$list\_f = mysql\_list\_fields($database,$table\_name);

// получаем список полей в таблице

$n = mysql\_num\_fields($list\_f); // число строк в результате

// предыдущего запроса (т.е. сколько всего

// полей в таблице Artifacts)

echo "<form method=post action=insert.php>";

// создаем форму для ввода данных

echo "&nbsp;<TABLE BORDER=0 CELLSPACING=0 width=50% ><tr>

<TD BGCOLOR='#005533' align=center><font color='#FFFFFF'>

<b> Add new row in $table\_name</b></font></td></tr><tr><td></td></tr></TABLE>";

echo "<table border=0 CELLSPACING=1 cellpadding=0 width=50% >";

// для каждого поля получаем его имя, тип, длину и флаги

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

$type = mysql\_field\_type($list\_f, $i);

$name\_f = mysql\_field\_name ($list\_f,$i);

$len = mysql\_field\_len($list\_f, $i);

$flags\_str = mysql\_field\_flags ($list\_f, $i);

// из строки флагов делаем массив,

// где каждый элемент массива - флаг поля

$flags = explode(" ", $flags\_str);

foreach ($flags as $f){

if ($f == 'auto\_increment') $key = $name\_f;

// запоминаем имя автоинкремента

}

/\* для каждого поля, не являющегося автоинкрементом, в

зависимости от его типа выводим подходящий элемент формы \*/

if ($key <> $name\_f){

echo "<tr><td align=right bgcolor='#C2E3B6'><font size=2>

<b>&nbsp;". $name\_f ."</b></font></td>";

switch ($type){

case "string":

$w = $len/5;

echo "<td><input type=text name=\"$name\_f\"

size = $w ></td>";

break;

case "int":

$w = $len/4;

echo "<td><input type=text name=\"$name\_f\"

size = $w ></td>";

break;

case "blob":

echo "<td><textarea rows=6 cols=60 name=\"$name\_f\"></textarea></td>";

break;

}

}

echo "</tr>";

}

echo "</table>";

echo "<input type=submit name='add' value='Add'>";

echo "</form>";

?>

Листинг 11.0.1. Форма для ввода данных в таблицу Artifacts

#### Запись данных в базу данных

Итак, форма создана. Теперь нужно сделать самое главное - отправить данные из этой формы в нашу базу данных. Как вы уже знаете, для того чтобы записать данные в таблицу, используется команда INSERT языка SQL. Например:

mysql> INSERT INTO Artifacts

SET title='Петров';

Возникает вопрос, как можно воспользоваться такой командой (или любой другой командой SQL) в PHP скрипте. Для этого существует функция *mysql\_query()*.

Синтаксис *mysql\_query*

ресурс mysql\_query ( строка query

[, ресурс link\_identifier])

*mysql\_query()* посылает SQL-запрос *активной базе данных* *MySQL* сервера, который определяется с помощью указателя link\_identifier(это ссылка на какое-то соединение с сервером *MySQL* ). Если параметр link\_identifier опущен, используется последнее открытое соединение. Если открытые соединения отсутствуют, функция пытается соединиться с СУБД, аналогично функции *mysql\_connect()* без параметров. Результат запроса буферизируется.

**Замечание**: *строка запроса* НЕ должна заканчиваться точкой с запятой.

Только для запросов SELECT, SHOW, *EXPLAIN*, *DESCRIBE*, *mysql\_query()* возвращает указатель на результат запроса, или FALSE, если запрос не был выполнен. В остальных случаях *mysql\_query()* возвращает TRUE, если запрос выполнен успешно, и FALSE - в случае ошибки. Значение, не равное FALSE, говорит о том, что запрос был выполнен успешно. Оно не говорит о количестве затронутых или возвращенных рядов. Вполне возможна ситуация, когда успешный запрос не затронет ни одного ряда. *mysql\_query()* также считается ошибочным и вернет FALSE, если у пользователя недостаточно прав для работы с указанной в запросе таблицей.

Итак, теперь мы знаем, как отправить запрос на вставку строк в базу данных. Заметим, что в предыдущем примере элементы формы мы назвали именами полей таблицы. Поэтому они будут доступны в скрипте insert.php, обрабатывающем данные формы, как переменные вида$*\_POST*['имя\_поля'].

<?

$conn=mysql\_connect("localhost","nina","123");// устанавливаем

// соединение

$database = "book";

$table\_name = "Artifacts";

mysql\_select\_db($database); // выбираем базу данных

$list\_f = mysql\_list\_fields($database,$table\_name);

// получаем список полей в таблице

$n = mysql\_num\_fields($list\_f); // число строк в результате

// предыдущего запроса

// составим один запрос сразу для всех полей таблицы

$sql = "INSERT INTO $table\_name SET "; // начинаем создавать

// запрос, перебираем все поля таблицы

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

$name\_f = mysql\_field\_name ($list\_f,$i); // вычисляем имя поля

$value = $\_POST[$name\_f]; // вычисляем значение поля

$j = $i + 1;

$sql = $sql . $name\_f." = '$value'"; // дописываем в

// строку $sql пару имя=значение

if ($j <> $n) $sql = $sql . ", "; // если поле не

// последнее в списке, то ставим запятую

}

// перед тем как записывать что-то в базу,

// можно посмотреть, какой запрос получился

//echo $sql;

$result = mysql\_query($sql,$conn); // отправляем запрос

// выводим сообщение успешно ли выполнен запрос

if (!$result) echo " Can't add ($table\_name) ";

else echo "Success!<br>";

?>

Листинг 11.0.2. insert.php

Итак, задачу добавления данных с помощью web-интерфейса мы решили. Однако тут есть одна тонкость. При решении мы не учитывали тот факт, что значения некоторых полей ( author, photo ) должны браться из других таблиц ( Persons, Images ). Поскольку *MySQL* с внешними ключами не работает (уже работает - прим. эксперта), этот момент остается на совести разработчиков системы, т.е. на нашей совести. Нужно дописать программу таким образом, чтобы была возможность вводить в такие поля правильные значения. Но мы делать этого не будем, поскольку задача лекции состоит в том, чтобы познакомить читателя с элементами технологии, а не в том, чтобы создать работающую систему. Кроме того, имеющихся у читателя знаний вполне достаточно, чтобы решить эту проблему самостоятельно. Мы же обратимся к другой задаче - отображение данных, хранящихся в базе данных СУБД *MySQL*.

### Отображение данных, хранящихся в MySQL

Чтобы отобразить какие-то данные в браузере с помощью *PHP*, нужно сначала получить эти данные в виде переменных *PHP*. При работе с *MySQL* без посредника (такого, как *PHP*) *выборка* данных производится с помощью команды SELECT языка *SQL*:

mysql> SELECT \* FROM Artifacts;

В предыдущей главе мы говорили, что любой *запрос*, в том числе и на выборку, можно отправить на *сервер* с помощью функции *mysql\_query()* ; Там у нас стояла немного другая задача - получить данные из формы и отправить их с помощью запроса на вставку в базу данных. Результатом работы *mysql\_query()* там могло быть только одно из выражений, TRUE или FALSE. Теперь же требуется отправить *запрос* на выбор всех полей, а результат отобразить в браузере. И здесь результат - это целая *таблица* значений, а точнее, *указатель* на эту таблицу. Так что нужны какие-то аналоги функции *mysql\_field\_name()*, только чтобы они извлекали из результата запроса не имя, а *значение* поля. Таких функций в *PHP* несколько. Наиболее популярные - *mysql\_result()* и *mysql\_fetch\_array()*.

*Синтаксис* *mysql\_result*

смешанное mysql\_result (ресурс result,

целое row [, смешанное field])

*mysql\_result()* возвращает *значение* одной ячейки результата запроса. *Аргумент* field может быть порядковым номером поля в результате, именем поля или именем поля с именем таблицы через точку tablename.*fieldname*. Если для имени поля в запросе применялся *алиас*('select foo as bar from...'), используйте его вместо реального имени поля.

Работая с большими результатами запросов, следует задействовать одну из функций, обрабатывающих сразу *целый* ряд результата (например, *mysql\_fetch\_row()*, *mysql\_fetch\_array()* и т.д.). Так как эти функции возвращают *значение* нескольких ячеек сразу, они НАМНОГО быстрее *mysql\_result()*. Кроме того, нужно учесть, что указание численного смещения (номера поля) работает намного быстрее, чем указание колонки или колонки и таблицы через точку.

Вызовы функции *mysql\_result()* не должны смешиваться с другими функциями, работающими с результатом запроса.

*Синтаксис* *mysql\_fetch\_array*

массив mysql\_fetch\_array ( ресурс result

[, целое result\_type])

Эта *функция* обрабатывает ряд результата запроса, возвращая *массив* (ассоциативный, численный или оба) с обработанным рядом результата запроса, или FALSE, если рядов больше нет.

***mysql\_fetch\_array()*** - это расширенная версия функции *mysql\_fetch\_row()*. Помимо хранения значений в массиве с численными индексами, функция возвращает значения в массиве с индексами по названию колонок.

Если несколько колонок в результате будут иметь одинаковые названия, будет возвращена последняя колонка. Чтобы получить *доступ* к первым, следует использовать численные *индексы массива* или *алиасы* в запросе. В случае *алиасов* именно их вы не сможете использовать в именах колонок, как, например, не сможете использовать "photo" в описанном ниже примере.

select Artifacts.photo as art\_image,

Persons.photo as pers\_image

from Artifacts, Persons

11.1. Запрос с дублирующимися именами колонок

Важно заметить, что *mysql\_fetch\_array()* работает НЕ медленнее, чем *mysql\_fetch\_row()*, и предоставляет более удобный *доступ* к данным.

Второй опциональный *аргумент* result\_type в функции *mysql\_fetch\_array()* является константой и может принимать следующие значения: MYSQL\_ASSOC, MYSQL\_NUM и MYSQL\_BOTH. Эта возможность добавлена в *PHP* 3.0.7. Значением по умолчанию является: MYSQL\_BOTH.

Используя MYSQL\_BOTH, получим *массив*, состоящий как из ассоциативных индексов, так и из численных. MYSQL\_ASSOC вернет только ассоциативные соответствия, а MYSQL\_NUM - только численные.

**Замечание**: имена полей, возвращаемые этой функцией, регистрозависимы.

Теперь отобразим данные из *Artifacts* в виде таблицы в браузере:

<?

/ \* сначала делаем то же, что и раньше: устанавливаем

соединение, выбираем базу и получаем список и число полей в таблице Artifacts \*/

$conn=mysql\_connect("localhost","nina","123");

$database = "book";

$table\_name = "Artifacts";

mysql\_select\_db($database);

$list\_f = mysql\_list\_fields($database,$table\_name);

$n1 = mysql\_num\_fields($list\_f);

// сохраним имена полей в массиве $names

for($j=0;$j<$n1; $j++){

$names[] = mysql\_field\_name ($list\_f,$j);

}

$sql = "SELECT \* FROM $table\_name"; // создаем SQL запрос

$q = mysql\_query($sql,$conn) or die(); // отправляем

// запрос на сервер

$n = mysql\_num\_rows($q); // получаем число строк результата

//рисуем HTML-таблицу

echo "&nbsp;<TABLE BORDER=0 CELLSPACING=0 width=90%

align=center><tr><TD BGCOLOR='#005533' align=center>

<font color='#FFFFFF'><b>$table\_name</b></font></td>

</tr></TABLE>";

echo "<table cellspacing=0 cellpadding=1 border=1

width=90% align=center>";

// отображаем названия полей

echo "<tr>";

foreach ($names as $val){

echo "<th ALIGN=CENTER BGCOLOR='#C2E3B6'>

<font size=2>$val</font></th>";

}

// отображаем значения полей

echo "</tr>";

for($i=0;$i<$n; $i++){ // перебираем все строки в

// результате запроса на выборку

echo "<tr>";

foreach ($names as $val) { // перебираем все

// имена полей

$value = mysql\_result($q,$i,$val); // получаем

// значение поля

echo "<td><font size=2>&nbsp;$value</font></td>";

// выводим значение поля

}

echo "</tr>";

}

echo "</table>";

Листинг 11.1.1. Отображение данных из Artifacts в виде таблицы в браузере

Сделаем то же самое с помощью *mysql\_fetch\_array()*:

<?

/\* ... начало то же, что и в предыдущем примере \*/

// отображаем значения полей

// получаем значение поля в виде ассоциативного массива

while($row = mysql\_fetch\_array($q, MYSQL\_ASSOC)) {

echo "<tr>";

foreach ($names as $val){

echo "<td><font size=2>&nbsp;$row[$val]</font></td>";

// выводим значение поля

}

echo "</tr>";

}

echo "</table>";

?>

Листинг 11.1.2. Отображение данных из Artifacts в виде таблицы в браузере. Вариант 2

### Заключение

В этой лекции мы решили две задачи: добавление данных в базу данных и их *отображение* в браузере с помощью языка *PHP*. Для этого мы рассмотрели ряд функций, которые позволяют отправлять *SQL*-запросы к базе данных и обрабатывать полученные ответы. Используя приведенную здесь технологию, можно решить *целый* ряд похожих задач, таких как задачи изменения и удаления данных, задачи манипулирования таблицами *базы данных* (т.е. их создание, изменение и удаление) и т.п. Все это типовые задачи, возникающие при разработке систем управления данными, и умение их решать, как и умение работать с базами данных в целом, очень важно для web-программиста.