

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Учреждение образования
«Гомельский государственный технический университет имени П.О. Сухого»
Кафедра «Информационные технологии»

Курс лекций
старшего преподавателя Стефановского И.Л.
по дисциплине

«Программирование в Internet»

для специальности 1–40 05 01 – "Информационные системы и технологии
(по направлениям)»,
направления специальности 1 40 05 01– 01 – «Информационные системы и
технологии (в проектировании и производстве)»

Гомель 2013

Содержание

Лекция 1 Введение. Виды динамических HTML-документов. CSS 3.0.....	3
Раздел 1. Программирование на стороне клиента. JavaScript	10
Лекция 2 Преимущества и ограничения программ, работающих на стороне клиента. Язык JavaScript. Основы синтаксиса.	10
Лекция 3 Объектная модель HTML страницы.....	16
Лекция 4 Событийная модель DHTML. Применение DHTML.....	23
Раздел 2. Программирование на стороне сервера.....	26
Лекция 5 Заголовки запроса и ответа. CGI. Переменные окружения	26
Лекция 6 Создание форм. Методы передачи параметров (GET, POST)	29
Лекция 7 HTTP - cookie. Атрибуты cookie. Сессии. Идентификация пользователя.	37
Раздел 3 Программирование на стороне сервера на PHP	39
Лекция 8 Введение в PHP.....	39
Лекция 9 Основы синтаксиса.....	41
Лекция 10 Управляющие конструкции.....	45
Лекция 11 Обработка запросов с помощью PHP	57
Лекция 12 Функции в PHP	63
Лекция 13 Объекты и классы в PHP.....	66
Лекция 14 Работа с массивами данных. Работа со строками	68
Лекция 15 Работа с файловой системой	76
Лекция 16 Авторизация доступа с помощью сессий.....	81
Лекция 17 Регулярные выражения	87
Лекция 18 Использование шаблонов в PHP	92
Список использованных источников	96

Лекция 1 Введение. Виды динамических HTML-документов. CSS 3.0

Динамический HTML

Динамический HTML (Dynamic HTML или DHTML) не является каким-то особым языком разметки страниц. Это всего лишь термин, применяемый для обозначения HTML-страниц с динамически изменяемым содержимым.

Реализация DHTML покоится на трех "китах": непосредственно HTML, каскадных таблицах стилей (Cascade Style Sheets — CSS) и языке сценариев (JavaScript или VBScript). Эти три компонента DHTML связаны между собой объектной моделью документа (Document Object Model — DOM), являющейся, по сути, интерфейсом прикладного программирования (API). DOM связывает воедино три перечисленных компонента, придавая простому документу HTML новое качество, — возможность динамического изменения своего содержимого без перезагрузки страницы.

Каскадные таблицы стилей можно сравнить со стилевыми файлами любого текстового редактора. С их помощью определяется внешний вид отображаемого HTML-документа: цвет шрифта и фона документа, сам шрифт, разбивка текста и многое другое. Для каждого элемента, задаваемого определенным тэгом HTML, можно определить свой стиль отображения в окне браузера. Например, заголовки первого уровня будут отображаться шрифтом Arial 16pt синего цвета, заголовки второго уровня — Arial 14pt красного цвета, основной текст — Times New Roman 10pt черного цвета с одинарным интервалом между строками. Можно создать таблицу стилей и использовать ее для всех документов, расположенных на сервере, что придаст стройность и строгость всему Web-сайту.

Объектная модель документа делает все элементы страницы программируемыми объектами. С ее помощью через языки сценариев можно получить доступ и управлять всем, что есть в документе. Каждый элемент HTML доступен как индивидуальный объект, а это означает, что можно изменять значение любого параметра любого тэга HTML-страницы, и, как следствие, документ действительно становится динамическим. Любое действие пользователя (щелчок кнопкой мыши, перемещение мыши в окне браузера или нажатие клавиши клавиатуры) объектной моделью документа трактуется как событие, которое может быть перехвачено и обработано процедурой сценария.

DHTML достаточно новая технология, и не все браузеры поддерживают объектную модель документа и каскадные таблицы стилей. Однако DHTML использует стандартные тэги HTML, и поэтому пользователи браузеров, не поддерживающих DOM, практически увидят все, что задумано разработчиком динамической страницы, но только в статическом виде.

Есть еще одна "неприятность", связанная с тем, что разные фирмы-разработчики браузеров могут реализовывать собственную объектную

модель документов, как это произошло с двумя популярными браузерами Internet Explorer и Netscape Navigator. Поэтому разработчикам динамических страниц приходится, в конечном счете, писать два варианта своих приложений, чтобы пользователи указанных браузеров могли правильно просматривать их страницы.

В данной главе описываются каскадные таблицы стилей, объектные модели документов браузеров Internet Explorer и Netscape Navigator и приемы создания динамических HTML-страниц.

Каскадные таблицы стилей

Каскадные таблицы стилей впервые были реализованы в Internet Explorer 3.0, но информация о них в то время была большей частью противоречивой. При реализации Internet Explorer 4.0 были приняты во внимание рекомендации REC-CSS1 Консорциума W3 относительно каскадных таблиц стилей, датированные 17 декабря 1996 года. К настоящему времени они пересмотрены и известны как рекомендации по каскадным таблицам стилей, уровень 1, документ REC-CSS1-19990111 от 11 января 1999 года. В мае 1998 года Консорциум издал рекомендации по каскадным таблицам стилей, уровень 2, документ REC-CSS2-19980512, часть из которых реализована в Internet Explorer 4.01. В соответствии с этими рекомендациями и будет вестись описание каскадных таблиц стилей.

Общие положения

Каскадные таблицы стилей, уровень 1, представляют собой простую технологию определения и присоединения стилей к документам HTML. Стил, говоря житейским языком, — это все то, что определяет внешний вид документа HTML при его отображении в окне браузера: шрифты и цвета заголовков разных уровней, шрифт и разрядка основного текста, задаваемого в тэге абзаца <P>, и т. д. Стил задается по определенным правилам, о которых, собственно говоря, и пойдет речь в данном разделе, а таблица стилей — набор правил отображения, применяемых в документе, к которому присоединена соответствующая таблица стилей.

Таблица стилей — это шаблон, который управляет форматированием тэгов HTML в Web-документе. Если читатель работал с текстовым редактором Microsoft Word, то концепция таблицы стилей напомнит ему концепцию стилевых файлов этого редактора: изменить внешний вид документа Word можно простым изменением присоединенных к нему стилей. Точно также изменить внешний вид документа HTML можно простым изменением присоединенной к нему таблицы стилей.

Почему в название таблиц стилей включено определение "каскадные"? Дело в том, что рекомендации Консорциума W3 позволяют использовать несколько таблиц стилей для управления форматированием одного документа HTML, а браузер по определенным правилам выстраивает приоритетность применения этих таблиц. Они выстраиваются неким "каскадом", по которому и "прокатывается" документ. Правила приоритетности и разрешения возникающих конфликтов описаны ниже в данном разделе.

Для разработки таблицы стилей достаточно немного ориентироваться в языке HTML и быть знакомым с базовой терминологией настольных издательских систем. Как отмечалось выше, таблица стилей представляет собой набор правил форматирования элементов HTML. Эти правила достаточно просты и легко запоминаемы. Например, если необходимо, чтобы в документе все заголовки первого уровня отображались синим цветом и шрифтом с кеглем (размером) 16 пунктов, то в таблице следует задать правило:

```
H1 {color: blue;  
font-size: 16pt}
```

Любое правило каскадных таблиц стилей состоит из двух частей: селектора и определения. Селектором может быть любой тэг HTML, для которого определение задает, каким образом необходимо его форматировать. Само определение, в свою очередь, также состоит из двух частей: свойства и его значения, разделенных знаком двоеточия (:). Назначение свойства очевидно из его названия. В приведенном правиле селектором является элемент H1, а определение, записанное в фигурных скобках, задает значения двух свойств заголовка первого уровня: цвет шрифта (свойство color) определен как синий (значение blue) и размер шрифта (свойство font-size) определен в 16 пунктов (значение 16pt). В одном правиле можно задать несколько определений, разделенных символом точка с запятой (;), как это демонстрируется в приведенном примере.

Созданная только что таблица стилей влияет на форматирование элемента определенного типа: заголовков первого уровня. Ее комбинация с другими таблицами стилей определяет окончательное представление документа при его просмотре в окне браузера.

Предупреждение

Синтаксис правил каскадных таблиц стилей не чувствителен к регистру. Селекторы, свойства и их значения можно задавать как строчными, так и прописными буквами, или в смешанном порядке. Однако каскадные таблицы стилей чувствительны к синтаксису задания правил и правильности названий свойств, значений и селекторов. Любая грамматическая ошибка приводит к тому, что правило пропускается анализатором браузера, и никакого предупреждающего сообщения не появляется.

Авторы страниц HTML должны писать свои таблицы стилей, только если они хотят придать документу вид, отличный от вида, предоставляемого умалчиваемой таблицей стилей браузера.

Встраивание таблиц стилей в документ

Чтобы таблица стилей могла воздействовать на внешнее представление документа, браузер должен знать о ее существовании. Для этого ее необходимо связать с HTML-документом.

Существует четыре способа связывания документа и таблицы стилей:

1. Связывание — позволяет использовать одну таблицу стилей для форматирования многих страниц HTML.

2. Внедрение — позволяет задавать все правила таблицы стилей непосредственно в самом документе.

3. импортирование — позволяет встраивать в документ таблицу стилей, расположенную на сервере.

4. Встраивание в тэги документа — позволяет изменять форматирование конкретных элементов страницы.

Связывание позволяет хранить таблицу стилей в отдельном файле и присоединять ее к документам с помощью тэга <LINK>, задаваемого в разделе

```
<HEAD>:
```

```
<LINK REL="stylesheet" TYPE="text/css" HREF="mystyles.css">
```

Связываемый файл содержит набор правил каскадных таблиц стилей, определяющих форматирование документа, и должен иметь расширение CSS.

Связывание позволяет разработчику применить одинаковый набор правил форматирования к группе HTML-документов, что приводит к единообразному отображению различных документов и придает некоторую системность серверу разработчика.

При внедрении таблицы стилей в документ правила, ее составляющие, задаются в стилевом блоке, ограниченном тэгами <STYLE TYPE="text/css"> и </STYLE>, который должен размещаться в разделе <HEAD> документа:

```
<HEAD>
```

```
<STYLE TYPE="text/css">
```

```
<!--
```

```
B { text-transform: uppercase; }
```

```
P { background-color: lightgrey;  
text-align:center; }
```

```
--->
```

```
</STYLE>
```

```
</HEAD>
```

Обычно браузеры, не поддерживающие какие-либо тэги, игнорируют их, интерпретируя, однако, их содержимое в том виде, в каком оно задано, что может приводить к ошибкам. Поэтому, как обычно, следует задавать содержимое тэгов, которые потенциально не обрабатываются старыми версиями браузеров, заключенным в тэг комментария <!-- ... -->.

В приведенном выше примере встроенная таблица стилей определяет отображение всех абзацев в документе (элемент P) на сером фоне с центрированными строками. Полужирный текст, определяемый любым элементом в (тэг) документа, будет отображаться прописными буквами, даже если в документе он задан строчными.

В тэге <STYLE> можно импортировать внешнюю таблицу стилей с помощью свойства @import таблицы стилей:

```
@import: url(mystyles.ess);
```

Его следует задавать в начале стилевого блока или связываемой таблицы стилей перед заданием остальных правил. Значением свойства @import является URL-адрес файла таблицы стилей.

Последний способ задания значений свойств таблицы стилей предназначен для оперативного форматирования определенного элемента документа и называется внедрение. Каждый тэг HTML имеет параметр STYLE, в котором можно задать значения его свойств в соответствии с синтаксисом каскадных таблиц стилей. Например, в следующем примере задается форматирование заголовка первого уровня, определяющее его отображение шрифтом красного цвета:

```
<H1 STYLE="color: red">Заголовок отображается шрифтом красного цвета</H1>
```

Если связанные, внедренные и импортируемые таблицы стилей влияют на форматирование всех элементов документа, для которых определены в таблицах правила, то встраивание определений стилей в конкретный тэг влияет на отображение только элемента, определяемого данным тэгом.

Совет

Рекомендуется избегать встраивания в тэги документа определений форматирования, так как подобная техника лишает разработчика преимуществ задания таблиц стилей в отдельном файле или в головной части документа, где их можно легко и быстро откорректировать, в случае необходимости. При форматировании отдельных тэгов придется просмотреть весь документ целиком, что требует достаточно большой и кропотливой работы.

Все способы встраивания таблиц стилей свободно сочетаются в одном документе. Например, можно разработать главную таблицу стилей для всех документов и связывать ее с каждым HTML-документом. импортируемая или внедряемая таблица стилей будет уточнять форматирование элементов конкретного документа, а встраиваемые в тэг определения форматирования будут уточнять их отображение.

Группирование и наследование

Правила каскадных таблиц стилей состоят из селектора и определения. Для уменьшения размеров таблиц стилей можно группировать разные селекторы в виде списка элементов страницы HTML, разделенных запятыми, если для них задается одно правило. Например, следующие правила

```
H1 {font-family: Arial}
```

```
H2 {font-family: Arial}
```

```
H3 {font-family: Arial}
```

можно сгруппировать и задать в виде одного правила со списком селекторов

```
H1, H2, H3 {font-family: Arial}
```

Аналогично группируются определения, только в списке они разделяются точками с запятой (;). Следующие правила форматирования заголовка первого уровня

```
H1 {font-weight: bold}
```

```
H1 {font-size: 14pt}
H1 {font-family: Arial}
```

можно задать в виде одного правила, сгруппировав определения:

```
H1 {font-weight: bold;
font-size: 14pt;
font-family: Arial;
}
```

Некоторые свойства имеют собственный синтаксис группирования, связанный с заданием значений нескольких свойств в одном. Например, предыдущий пример при использовании свойства `font` запишется так:

```
H1 {font: bold 14pt Arial}
```

При задании таблицы стилей можно свободно комбинировать все три правила группирования для уменьшения ее размеров.

В HTML некоторые элементы могут содержать другие. Как будет отображаться элемент, расположенный внутри другого элемента страницы, если для последнего задано правило форматирования, а для вложенного элемента нет? Например, пусть цвет шрифта абзаца определен как синий (`color: blue`). Как будет отображаться выделенный элемент текста, задаваемый тэгом ``, если для него не определено правило форматирования? В подобных случаях вложенный элемент наследует правила форматирования элемента-родителя. В нашем примере выделенный элемент будет также отображаться синим цветом. Другие свойства ведут себя аналогично свойству `color`, например `font-family`, `font-size`.

Некоторые свойства не наследуются вложенными элементами от своих родителей, например свойство `background`, но по умолчанию вложенные элементы будут отображаться с фоном родительского элемента.

Наследование полезно при задании значений свойств, применяемых к документу по умолчанию. Для этого достаточно задать все свойства для элемента, порождающего все остальные элементы страницы HTML. Таким элементом является тело документа, определяемое тэгом `<BODY>` :

```
BODY {
color: black;
font-family: "Times New Roman";
background: url(texture.gif) white;
}
```

Приведенные правила задают форматирование документа по умолчанию: черным шрифтом гарнитуры Times New Roman с фоном, задаваемым графическим файлом `texture.gif`, или на белом фоне, если файл не доступен.

Примечание

Приведенное задание правил форматирования по умолчанию будет работать всегда, даже если разработчик пропустит в документе тэг `<BODY>`, что допускается стандартом языка HTML, так как синтаксический анализатор HTML всегда вставляет пропущенный тэг `<BODY>`.

Селекторы

Правила каскадных таблиц стилей, в которых в качестве селектора используются тэги HTML, влияют на отображение всех элементов заданного типа в документе. Следующее правило отображает без подчеркивания все ссылки (тэг <A>) в документе:

```
<STYLE TYPE="text/css">
<!--
A { text-decoration:none; }
-->
</STYLE>
```

А что делать, если необходимо некоторые ссылки отобразить по-другому? Можно задать для них правило форматирования непосредственно в тэге, а можно применить параметр CLASS, добавленный в HTML 4.0 в качестве стандарта для всех тэгов. Значением параметра CLASS является ссылка на класс, задаваемый в таблице стилей.

Что такое CSS3?

CSS3 - это новый стандарт оформления HTML документов значительно расширяющий возможности предыдущего стандарта CSS2.1.

Многие возможности, которые были труднодоступны в CSS2.1, то есть требовали использования дополнительных внешних программ (таких как Adobe Photoshop), скриптов (таких как JavaScript) или специальных "хитростей" могут легко достигаться в CSS3 за счет использования новых свойств оформления.

В CSS3 Вы можете:

- Создавать элементы со сглаженными углами;
- Создавать линейные и сферические градиенты;
- Более гибко оформлять фоновую картинку элементов;
- Добавлять к элементам и к тексту элементов тени;
- Использовать небезопасные шрифты (не боясь при этом, что они будут не поддерживаться браузером пользователя);
- Создавать анимацию и различные эффекты переходов;

Задавать цвета несколькими новыми способами и многое другое.

Пример:

```
background:-webkit-linear-gradient(top,#E567B1,#84004D);
background:-moz-linear-gradient(top,#CB0077,black);
background:-o-linear-gradient(top,#CB0077,black);
box-shadow:3px 3px 10px 1px #000000;
```

Обратите внимание: новые CSS3 свойства разобранные в данном учебнике поддерживаются только в современных браузерах: IE9+, Firefox 3.6+, Opera 10+, Chrome 12+, Safari 5+. Особые случаи будут специально оговариваться.

Обратите внимание: спецификация CSS3 от W3C все еще находится в разработке, поэтому поведение некоторых свойств рассмотренных в данном учебнике может измениться.

Раздел 1. Программирование на стороне клиента. JavaScript

Лекция 2 Преимущества и ограничения программ, работающих на стороне клиента. Язык JavaScript. Основы синтаксиса.

Что такое JavaScript

JavaScript - новый язык для составления скриптов, разработанный фирмой Netscape. С помощью JavaScript Вы можете легко создавать интерактивные Web-страницы. В данном руководстве Вы увидите, что можно сделать с помощью JavaScript, и даже более того - увидите, как это сделано.

Запуск JavaScript

Что необходимо сделать, чтобы запускать скрипты, написанные на языке JavaScript? Вам понадобится браузер, способный работать с JavaScript - например Netscape Navigator (начиная с версии 2.0) или Microsoft Internet Explorer (MSIE - начиная с версии 3.0). С тех пор, как оба этих браузера стали широко распространены, множество людей получили возможность работать со скриптами, написанными на языке JavaScript. Несомненно, это важный аргумент в пользу выбора языка JavaScript, как средства улучшения ваших Web-страниц.

При этом, возможно, Вы обнаружите, что много хороших средств диалога можно создать, пользуясь лишь командами HTML. Чтобы получить дополнительную информацию о языке HTML, лучше всего инициировать поиск по ключевому слову 'html' на поисковом сервере Yahoo.

Размещение JavaScript на HTML-странице

Код скрипта JavaScript размещается непосредственно на HTML-странице. Чтобы увидеть, как делается, давайте рассмотрим следующий простой пример:

```
<html>
<body>
<br>
Это обычный HTML документ.
<br>
  <script language="JavaScript">
    document.write("А это JavaScript!")
  </script>
<br>
Вновь документ HTML.
</body>
</html>
```

С первого взгляда пример напоминает обычный файл HTML. Единственное новшество здесь - конструкция:

```
<script language="JavaScript">
  document.write("А это JavaScript!")
</script>
```

Это действительно код JavaScript. Чтобы видеть, как этот скрипт работает, запишите данный пример как обычный файл HTML и загрузите его в браузер, имеющий поддержку языка JavaScript. В результате Вы получите 3 строки текста:

```
Это обычный HTML документ.
А это JavaScript!
Вновь документ HTML.
```

Я должен признать, что данный скрипт не столь полезен - то же самое и более просто можно было бы написать на "чистом" языке HTML. Я всего лишь хотел продемонстрировать Вам тэг признака `<script>`. Все, что стоит между тэгами `<script>` и `</script>`, интерпретируется как код на языке JavaScript. Здесь Вы также видите пример использования инструкции `document.write()` - одной из наиболее важных команд, используемых при программировании на языке JavaScript. Команда `document.write()` используется, когда необходимо что-либо написать в текущем документе (в данном случае таким является наш HTML-документ). Так наша небольшая программа на JavaScript в HTML-документе пишет фразу "А это JavaScript!".

Браузеры без поддержки JavaScript

А как будет выглядеть наша страница, если браузер не воспринимает JavaScript? Браузеры, не имеющие поддержки JavaScript, "не знают" и тэга `<script>`. Они игнорируют его и печатают все стоящие вслед за ним коды как обычный текст. Иными словами, читатель увидит, как код JavaScript, приведенный в нашей программе, окажется вписан открытым текстом прямо посреди HTML-документа. Разумеется, это не входило в наши намерения. На этот случай имеется специальный способ скрыть исходный код скрипта от старых версий браузеров - мы будем использовать для этого тэг комментария из HTML - `<!-- -->`. В результате новый вариант нашего исходного кода будет выглядеть как:

```
<html>
<body>
<br>
Это обычный HTML документ.
<br>
  <script language="JavaScript">
    <!-- скрывает код от старых браузеров

    document.write("А это JavaScript!")

  // -->
```

```
</script>
<br>
Вновь документ HTML.
</body>
</html>
```

В этом случае браузер без поддержки JavaScript будет печатать:

Это обычный HTML документ.
Вновь документ HTML.

А без HTML-тэга комментария браузер без поддержки JavaScript напечатал бы:

Это обычный HTML документ.
document.write("А это JavaScript!")
Вновь документ HTML.

Пожалуйста обратите внимание, что Вы не можете полностью скрыть исходный код JavaScript. То, что мы здесь делаем, имеет целью предотвратить распечатку кода скрипта на старых браузерах - однако тем не менее читатель сможет увидеть этот код посредством пункта меню 'View document source'. Не существует также способа скрыть что-либо от просмотра в вашем исходном коде (и увидеть, как выполнен тот или иной трюк).

События

События и обработчики событий являются очень важной частью для программирования на языке JavaScript. События, главным образом, инициируются теми или иными действиями пользователя. Если он щелкает по некоторой кнопке, происходит событие "Click". Если указатель мыши пересекает какую-либо ссылку гипертекста - происходит событие MouseOver. Существует несколько различных типов событий. Мы можем заставить нашу JavaScript-программу реагировать на некоторые из них. И это может быть выполнено с помощью специальных программ обработки событий. Так, в результате щелчка по кнопке может создаваться выпадающее окно. Это означает, что создание окна должно быть реакцией на событие щелчка - Click. Программа - обработчик событий, которую мы должны использовать в данном случае, называется onClick. И она сообщает компьютеру, что нужно делать, если произойдет данное событие. Приведенный ниже код представляет простой пример программы обработки события onClick:

```
<form>
<input type="button" value="Click me" onClick="alert('Yo')">
</form>
```

Данный пример имеет несколько новых особенностей - рассмотрим их по порядку. Вы можете здесь видеть, что мы создаем некую форму с кнопкой (как это делать - проблема языка HTML, так что рассматривать это здесь я не буду). Первая новая особенность - `onClick="alert('Yo')"` в тэге `<input>`. Как мы уже говорили, этот атрибут определяет, что происходит, когда нажимают на кнопку. Таким образом, если имеет место событие Click, компьютеру должен выполнить вызов `alert('Yo')`. Это и есть пример кода на языке JavaScript (Обратите внимание, что в этом случае мы даже не пользуемся тэгом `<script>`). Функция `alert()` позволяет Вам создавать выпадающие окна. При ее вызове Вы должны в скобках задать некую строку. В нашем случае это 'Yo'. И это как раз будет тот текст, что появится в выпадающем окне. Таким образом, когда читатель когда щелкает на кнопке, наш скрипт создает окно, содержащее текст 'Yo'.

Некоторое замешательство может вызвать еще одна особенность данного примера: в команде `document.write ()` мы использовали двойные кавычки (`"`), а в конструкции `alert()` - только одинарные. Почему? В большинстве случаев Вы можете использовать оба типа кавычек. Однако в последнем примере мы написали `onClick="alert('Yo')"` - то есть мы использовали и двойные, и одинарные кавычки. Если бы мы написали `onClick="alert("Yo")"`, то компьютер не смог бы разобраться в нашем скрипте, поскольку становится неясно, к которой из частей конструкции имеет отношение функция обработки событий `onClick`, а к которой - нет. Поэтому Вы и вынуждены в данном случае перемежать оба типа кавычек. Не имеет значения, в каком порядке Вы использовали кавычки - сперва двойные, а затем одинарные или наоборот. То есть Вы можете точно так же написать и `onClick='alert("Yo")'`.

Вы можете использовать в скрипте множество различных типов функций обработки событий. Сведения о некоторых из них мы получим в данном описании, однако не о всех. Поэтому обращайтесь пожалуйста к соответствующему справочнику, если Вы хотите узнать, какие обработчики событий еще существуют.

Итак, если Вы используете браузер Netscape Navigator, то выпадающее окно содержит текст, что был передан функции JavaScript `alert`. Такое ограничение накладывается по соображениям безопасности. Такое же выпадающее окно Вы можете создать и с помощью метода `prompt()`. Однако в этом случае окно будет воспроизводить текст, введенный читателем. А потому, скрипт, написанный злоумышленником, может принять вид системного сообщения и попросить читателя ввести некий пароль. А если текст помещается в выпадающее окно, то тем самым читателю дается понять, что данное окно было создано web-браузером, а не вашей операционной системой. И поскольку данное ограничение наложено по соображениям безопасности, Вы не можете взять и просто так удалить появившееся сообщение.

Функции

В большинстве наших программ на языке JavaScript мы будем пользоваться функциями. Поэтому уже теперь мне необходимо рассказать об этом важном элементе языка. В большинстве случаев функции представляют собой лишь способ связать вместе нескольких команд. Давайте, к примеру, напишем скрипт, печатающий некий текст три раза подряд. Для начала рассмотрим простой подход:

```
<html>
<script language="JavaScript">
<!-- hide
document.write("Добро пожаловать на мою страницу!<br>");
document.write("Это JavaScript!<br>");
document.write("Добро пожаловать на мою страницу!<br>");
document.write("Это JavaScript!<br>");
document.write("Добро пожаловать на мою страницу!<br>");
document.write("Это JavaScript!<br>");
// -->
</script>
</html>
```

И такой скрипт напишет следующий текст:

Добро пожаловать на мою страницу!

Это JavaScript!

три раза. Если посмотреть на исходный код скрипта, то видно, что для получения необходимого результата определенная часть его кода была повторена три раза. Разве это эффективно? Нет, мы можем решить ту же задачу еще лучше. Как насчет такого скрипта для решения той же самой задачи?:

```
<html>
<script language="JavaScript">
<!-- hide
function myFunction() {
    document.write("Добро пожаловать на мою страницу!<br>");
    document.write("Это JavaScript!<br>");
}
myFunction();
myFunction();
myFunction();
// -->
</script>
</html>
```

В этом скрипте мы определили некую функцию, состоящую из следующих строк:

```
function myFunction() {
    document.write("Добро пожаловать на мою страницу!<br>");
    document.write("Это JavaScript!<br>");
}
```

Все команды скрипта, что находятся внутри фигурных скобок - {} - принадлежат функции myFunction (). Это означает, что обе команды document.write() теперь связаны воедино и могут быть выполнены при вызове указанной функции. И действительно, нашем примере есть три вызова этой функции - Можно увидеть, что мы написали строку myFunction() три раза сразу после того, как дали определение самой функции. То есть как раз и сделали три вызова. В свою очередь, это означает, что содержимое этой функции (команды, указанные в фигурных скобках) было выполнено трижды. Поскольку это довольно простой пример использования функции, то у Вас мог возникнуть вопрос, а почему собственно эти функции столь важны в JavaScript. По прочтении данного описания Вы конечно же поймете их пользу. Именно возможность передачи переменных при вызове функции придает нашим скриптам подлинную гибкость - что это такое, мы увидим позже.

Функции могут также использоваться совместно с процедурами обработки событий. Рассмотрим следующий пример:

```
<html>
<head>
<script language="JavaScript">
<!-- hide
function calculation() {
    var x= 12;
    var y= 5;
    var result= x + y;
    alert(result);
}
// -->
</script>
</head>
<body>
<form>
<input type="button" value="Calculate" onClick="calculation()">
</form>
</body>
</html>
```

Здесь при нажатии на кнопку осуществляется вызов функции calculation(). Как можно заметить, эта функция выполняет некие вычисления, пользуясь переменными x, y и result. Переменную мы можем определить с помощью ключевого слова var. Переменные могут использоваться для хранения различных величин - чисел, строк текста и т.д. Так строка скрипта var result= x + y; сообщает браузеру о том, что необходимо создать

переменную `result` и поместить туда результат выполнения арифметической операции $x + y$ (т.е. $5 + 12$). После этого в переменный `result` будет размещено число 17. В данном случае команда `alert(result)` выполняет то же самое, что и `alert(17)`. Иными словами, мы получаем выпадающее окно, в котором написано число 17.

Лекция 3 Объектная модель HTML страницы

Объектная модель документа (Document Object Model – DOM) является стандартом, предложенным веб-консорциумом, и регламентирует способ представления содержимого документа (в частности веб-страницы) в виде набора объектов. Под содержимым понимается все, что может находиться на веб-странице: рисунки, ссылки, абзацы, текст и т. д.

В отличие от объектной модели браузера (BOM), которая уникальна для каждого браузера, объектная модель документа является стандартом и должна поддерживаться всеми браузерами. И хотя на практике поддержка DOM реализована не в полной мере, тем не менее необходимо стремиться следовать требованиям этого стандарта как производителям браузеров, так и разработчикам веб-сайтов.

Следует заметить, что DOM может применяться не только в веб-страницах, но и к любым другим документам. В частности, она может использоваться с любыми словарями XML, причем одним из таких словарей является HTML, а точнее, XHTML.

DOM является развивающимся стандартом и разбит на три уровня. Первый уровень является первой версией стандарта и пока что единственной законченной. Он состоит из двух разделов: первый является ядром и определяет принципы манипуляции со структурой документа (генерация и навигация), а второй посвящен представлению в DOM элементов HTML, определяемых одноименными тегами.

Второй и третий уровни описывают модель событий, дополняют таблицы стилей, проходы по структуре.

Представление документа в виде древовидной структуры

В DOM документ представляется в виде древовидной структуры (рис. 3.1), являющейся одной из наиболее употребительных структур в программировании. Это обеспечивает унифицированный способ навигации по документу.


```

<html>
<head>
  <title>Пример представления HTML-документа в виде дерева</title>
</head>
<body>
  <h1>Представление документа в виде дерева</h1>
  <p1>Абзац 1</p>
  <p1>Абзац 1</p>
</body>
</html>

```



Рис. 3.1. Представление HTML-документа в виде древовидной структуры

Навигация по документу

В модели DOM к элементу можно обратиться непосредственно по его идентификатору id, воспользовавшись методом `getElementById` объекта `Document`:

```

<html>
<head>
  <title>Основы DOM</title>
</head>
<body>
  <h1 id = "head">Основы DOM</h1>
  <p>A Text</p>
  <script language = "JavaScript">

```

```

var a = document.getElementById("head");
alert(a);
</script>
</body>
</html>

```

Для получения коллекции всех элементов, соответствующих какому-либо тегу, используется метод объекта Document – `getElementsByTagName`. Например, `var a = document.getElementsByTagName("TD")` присвоит переменной `a` коллекцию всех элементов `<td>`. Обратите внимание, что имя элемента следует писать прописными буквами ("TD"). Рассмотрим пример использования метода `getElementsByTagName`:

```

<html>
<head>
<title>Основы DOM</title>
</head>
<body>
<h1 id = "head">Основы DOM</h1>
<table border = "2">
<tr>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>2,1</td>
<td>2,2</td>
</tr>
</table>
<script language = "JavaScript">
var a = document.getElementsByTagName("TD");
a.item(0).style.color = "red";
a.item(3).style.fontFamily = "arial";
a.item(3).style.color = "green";
</script>
</body>
</html>

```

Чтобы воспользоваться преимуществом древовидной структуры, принятой в DOM для представления документа, следует использовать навигационные атрибуты (рис. 3.2), представленные в табл. 3.1.

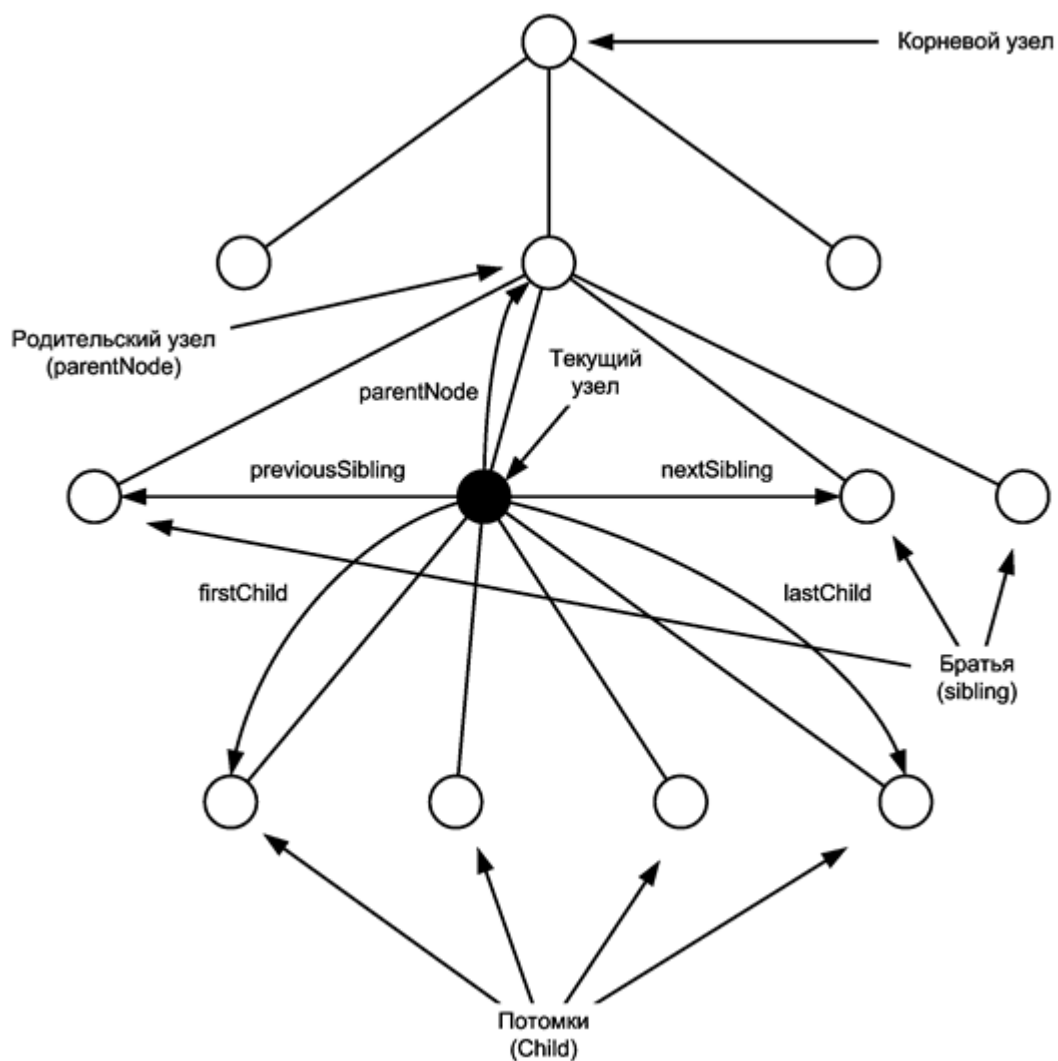


Рис. 3.2. Навигационные атрибуты объекта Node

Таблица 3.1 Навигационные атрибуты объекта Node

Атрибут	Описание
firstChild	Возвращает первый узел-потомок
lastChild	Возвращает последний узел-потомок
previousSibling	Возвращает предыдущий соседний узел, имеющий с текущим одного родителя
nextSibling	Возвращает следующий соседний узел, имеющий с текущим одного родителя
parentNode	Возвращает родительский узел
ownerDocument	Возвращает корневой узел документа, содержащий текущий узел
nodeName	Возвращает имя узла
nodeValue	Возвращает значение узла в текстовом формате

nodeType	Возвращает тип узла в виде числа
----------	----------------------------------

В следующем примере осуществляется проход по древовидной структуре документа:

```

<html>
<head>
<title>Навигация по документу</title>
</head>
<body>
<h1>Изучение навигации по документу</h1>
<p>Абзац 1</p>
<p>Абзац 2</p>
<script language = "JavaScript">
var temp = document.documentElement;
temp = temp.firstChild;
alert(temp.tagName);
if(temp.nextSibling == 3)
temp = temp.nextSibling.nextSibling;
else
temp = temp.nextSibling;
alert(temp.tagName);
temp = temp.firstChild;
alert(temp.tagName);
temp.style.color = "red";
if(temp.nextSibling == 3)
temp = temp.nextSibling.nextSibling;
else
temp = temp.nextSibling;
alert(temp.tagName);
temp.style.color = "blue";
temp = temp.parentNode;
alert(temp.tagName);
</script>
</body>
</html>

```

Динамическая генерация веб-страниц средствами DHTML на основе DOM.

Иногда требуется динамически формировать веб-страницы, например, в случае создания чатов, форумов, либо динамически создаваемых веб-страниц, содержимое которых хранится в базе данных. DOM позволяет решить такую задачу.

Для создания объектов у объекта Document имеются следующие методы (табл. 3.2):

Таблица 3.2 Методы объекта Document, позволяющие создавать объекты

Метод	Описание
createElement(имя_элемента)	Создает новый узел элемента с указанным именем
createTextNode(текст)	Создает текстовый узел с указанным текстом
createAttribute(имя_атрибута)	Создает новый узел атрибута с указанным именем

Вновь созданные объекты добавляются в структуру документа при помощи методов объекта Node (табл. 3.3):

Таблица 3.3 Методы объекта Node, добавляющие и удаляющие элементы документа

Метод	Описание
appendChild(новый_узел)	Добавляет объект Node в конец списка узлов-потомков
cloneNode(потомок-опция)	Создает объект Node, идентичный указанному в аргументе. В качестве аргумента можно использовать и все узлы-потомки одновременно
hasChildNodes()	Возвращает true, если узел имеет потомков
insertBefore(новый_узел, текущий_узел)	Вставляет объект Node в список потомков перед узлом, указанным в качестве второго параметра
removeChild(узел-потомок)	Удаляет узел-потомок, указанный в качестве параметра
replaceChild(новый_потомок, старый_потомок)	Заменяет старого потомка на нового

Приведем пример динамической генерации документа средствами DOM (рис. 3.3):

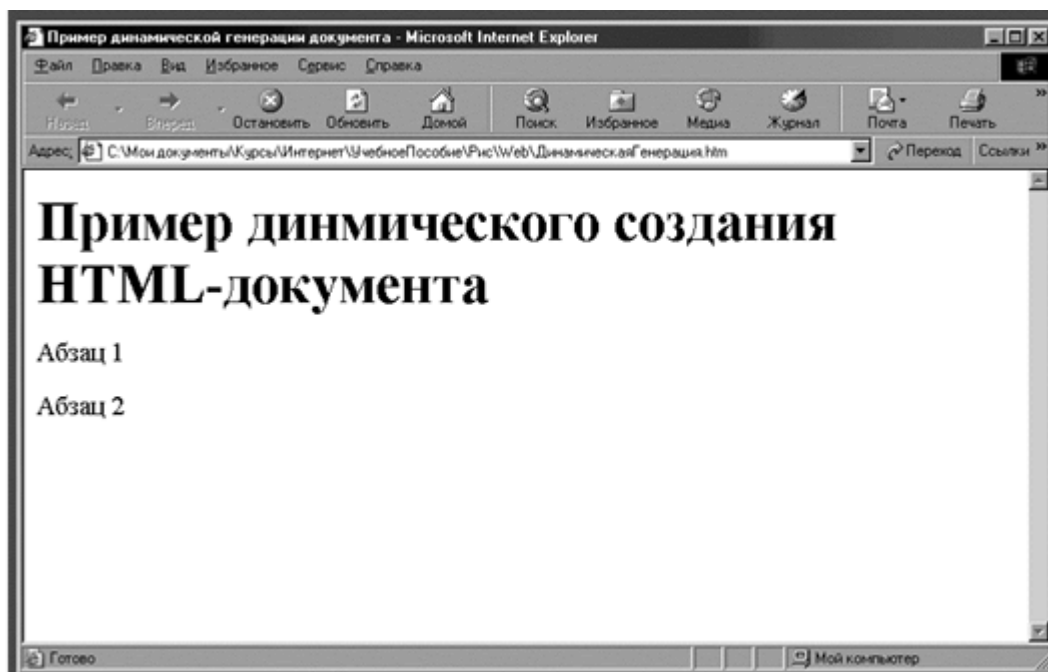


Рис. 3.3. Динамически сгенерированная веб-страница

```
<html>
<head>
<title>Пример динамической генерации документа</title>
</head>
<body>
<script language = "JavaScript">
var newText;
var newElem;
newText = document.createTextNode("Пример
динмического создания HTML-документа");
newElem = document.createElement("H1");
newElem.appendChild(newText);
document.body.appendChild(newElem);
newText = document.createTextNode("Абзац");
newElem = document.createElement("P");
newElem.appendChild(newText);
document.body.appendChild(newElem);
</script>
</body>
</html>
```

Для чтения и установки атрибутов используются следующие методы объекта Element (табл. 3.4).

Таблица 3.4 Методы объекта Element

Метод	Описание
getAttribute(имя_атрибута)	Возвращает значение атрибута

setAttribute(имя_атрибута, значение)	Устанавливает значение атрибута
removeAttribute(имя_атрибута)	Устанавливает значение атрибута по умолчанию, затирая текущее значение

Ниже приведен пример на задание атрибутов. И хотя применение атрибутов физического форматирования не рекомендовано к применению (для этих целей используются стили), они были выбраны в качестве примера, так как наглядно демонстрируют идею задания атрибутов методами DOM.

```

<html>
<head>
<title>Пример динамического создания HTML-документа</title>
</head>
<body>
<script language = "JavaScript">
var newText;
var newElem;
newText = document.createTextNode("Пример динамического
создания HTML-документа");
newElem = document.createElement("H1");
newElem.appendChild(newText);
newElem.setAttribute("align", "center");
document.body.appendChild(newElem);
alert(newElem.getAttribute("align"));
newText = document.createTextNode("Абзац");
newElem = document.createElement("P");
newElem.appendChild(newText);
newElem.setAttribute("align", "right");
document.body.appendChild(newElem);
alert(newElem.getAttribute("align"));
newElem.removeAttribute("align");
</script>
</body>
</html>

```

Лекция 4 Событийная модель DHTML. Применение DHTML

Особенностью программ, создаваемых для среды веб является то, что они управляются событиями. Чтобы узнать, какое событие произошло, в DOM имеется объект события event(табл. 4.1). Объект event является локальным и его следует явным образом передавать в обработчик события.

Таблица 4.1 Свойства объекта event

Свойство	Описание
----------	----------

bubbles	Указывает возможность «всплывания» события (передачи управления вверх по иерархической структуре)
cancelable	Указывает возможность отмены действия события, заданного по умолчанию
currentTarget	Указывает событие, обрабатываемое в данный момент
eventPhase	Указывает фазу возбуждения события
target (только NN 6)	Указывает элемент, вызвавший событие
timestamp (только NN 6)	Указывает время возникновения события
type	Указывает имя события

Приведем пример динамически изменяемого текста.

```

<html>
<head>
<script language = "JavaScript">
function onMouseover()
{
var temp = document.getElementById("DynamicText");
temp.firstChild.nodeValue = "Указатель мыши на тексте";
temp.style.color = "red";
}
function onMouseout()
{
var temp = document.getElementById("DynamicText");
temp.firstChild.nodeValue = "Динамический текст";
temp.style.color = "black";
}
</script>
<title>Динамический текст</title>
</head>
<body>
<p id = "DynamicText" onmouseover = "return onMouseover()"
onmouseout = "return onMouseout()">Динамический текст</p>
</body>
</html>

```

Таблица 4.2 Свойства объекта mouse

Свойство	Описание
----------	----------

altKey	Указывает, была ли нажата клавиша <Alt> в момент возникновения события
button	Указывает, какая клавиша мыши была нажата
clientX	Сообщает горизонтальную координату указателя мыши в окне браузера на момент возникновения события
clientY	Сообщает вертикальную координату указателя мыши в окне браузера на момент возникновения события
ctrlKey	Указывает, была ли нажата клавиша <Ctrl> в момент возникновения события
metaKey	Указывает, была ли нажата метаклавиша в момент возникновения события
relatedTarget (только NN 6)	Указывает цель события
screenX	Сообщает горизонтальную координату указателя мыши в окне браузера, вычисленную от начала экранных координат, на момент возникновения события
screenY	Сообщает вертикальную координату указателя мыши в окне браузера, вычисленную от начала экранных координат, на момент возникновения события
shiftKey	Указывает, была ли нажата клавиша <Shift> в момент возникновения события

Пример, определяющий координаты нахождения курсора мыши:

```

<html>
<head>
<title>Определение координат курсора</title>
<script language = "JavaScript">
function onClick(e)
{
alert("X = " + e.clientX + "; " + "Y = " + e.clientY);
}
</script>
</head>
<body onclick = "onClick(event)">
<p>Щелкните мышью на экране<p>
</body>
</html>

```

Раздел 2. Программирование на стороне сервера

Лекция 5 Заголовки запроса и ответа. CGI. Переменные окружения

CGI

CGI – (Common Gateway Interface – Общий интерфейс маршрутизации) служит для обеспечения связи внешней прикладной программы с Web-сервером. Обычно Web-серверы возвращают статическую информацию, то есть вы набираете некоторый URL (в своем браузере) в котором задаете имя ресурса, который вы хотите получить. Web-сервер анализирует ваш запрос и отправляет вам то что вы там хотели (или сообщение об ошибке, если указанный вами ресурс недоступен) вот вообще-то и весь World Wide Web. Во всем этом есть одна проблема информация которую посылает сервер по своей природе статична. Сервер не может вам послать больше того что у него есть на момент вашего запроса. Вот чтобы изменить эту ситуацию к лучшему и был введен стандарт CGI. CGI – расширяет возможности Web-сервера, тем что информация поступающая от Web-сервера приобретает динамический характер. Делается это следующим образом: Если в URL (в браузере) указать не какой-нибудь статический ресурс, а специальную программу (CGI-скрипт). А сервер проанализировав такой запрос возьмет и запустит ее. А она в свою очередь (пользуясь всеми ресурсами сервера) выдаст некоторую динамическую информацию (например: страничку). А Web-сервер после окончания работы CGI-скрипта отправит эту информацию браузеру предварительно снабдив ее нужным для протокола HTTP заголовком.

Например: Есть CGI-скрипт который обрабатывает данные с отосланной вами формы и заносит их в серверную базу данных, а вам генерирует ответ (в виде HTML-страницы), что вы большой молодец или сообщение об ошибке если что-то не в порядке с вашими данными.

Так вот CGI – и является именно тем стандартом который описывает как сервер должен запускать CGI-скрипт, как должен передавать ему параметры HTTP-запроса и как CGI-скрипт должен передавать результаты своей работы серверу. Вот об этом мы и поговорим.

CGI-скрипт – это программа поддерживающая интерфейс CGI, она может быть написана на любом языке программирования (Perl, C/C++, Fortran, Unix Shell, Visual Basic, Delphi, Pascal, Apple Script, Java, Oberon, ... на любом) все будет зависеть от сервера на котором будет работать данная программа. Как выполнимый модуль она обычно располагается в директории /cgi-bin сервера или в директории /scripts (на самом деле это зависит от настроек Web-сервера, просто ему надо знать из какой директории надо запускать программы, а из какой пересылать находящиеся там данные). Существуют также такие CGI-скрипты которые взаимодействуют с ППО (прикладным программным обеспечением) сервера (СУБД, электронные таблицы, деловая графика), а по результатам взаимодействия создают ответ и

передают его серверу, который в свою очередь отсылает его пользователю. Такие CGI-скрипты называются шлюзами они выполняют роль переходника между пользователем и ППО сервера.

Ну а теперь рассмотрим структуру CGI. Во-первых рассмотрим каким-образом сервер передает CGI-скрипту информацию об HTTP-запросе. А делает он это следующим образом: Перед запуском CGI-скрипта Web-сервер создает для него Переменные окружения (Environment variables) или как их еще называют переменные среды и записывает в них всю информацию о текущем запросе. Перечислим их:

Переменные окружения CGI.

REQUEST_METHOD – Это одна из самых главных переменных используемая для определения метода запроса HTTP.

Пример:REQUEST_METHOD=GET

QUERY_STRING - Это строка запроса при методе GET.

Пример:QUERY_STRING= name=Vasya&age=19&hobby=games

CONTENT_LENGTH - Длина в байтах тела запроса. Соответствует полю Content-Length в протоколе HTTP.

Пример:CONTENT_LENGTH=31

CONTENT_TYPE - MIME тип тела запроса (для форм кодированных выше указанным образом он application/x-www-form-urlencoded). Соответствует полю Content-Type в HTTP протоколе

GATEWAY_INTERFACE - Версия протокола CGI.

Пример:GATEWAY_INTERFACE=CGI/1.1

REMOTE_ADDR - IP-Адрес клиента , делающего данный запрос.

Пример:REMOTE_ADDR=192.168.3.1

REMOTE_HOST - Если запрашивающий клиент имеет доменное имя, то эта переменная содержит его, в противном случае - тот же самый IP-адрес что и в REMOTE_ADDR

Пример:REMOTE_HOST=uni-vologda.ac.ru

SCRIPT_NAME - Имя скрипта, использованное в запросе. Для получения реального пути на сервере используйте SCRIPT_FILENAME.

Пример:SCRIPT_NAME=/~rva/guestbook.cgi

SCRIPT_FILENAME - Имя файла скрипта на сервере.

Пример:SCRIPT_FILENAME=/students/rva/cgi-bin/guestbook.cgi

SERVER_NAME - Имя сервера ,чаще всего доменное как www.microsoft.com ,но в редких случаях за неимением такового может быть IP-адресом как 192.168.3.1

Пример:SERVER_NAME=www.vologda.ru

SERVER_PORT - TCP-Порт сервера использующийся для соединения . По умолчанию HTTP-порт 80, хотя может быть в некоторых случаях быть другим.

Пример:SERVER_PORT=80

SERVER_PROTOCOL - Версия протокола сервера.

Пример:SERVER_PROTOCOL=HTTP/1.1

SERVER_SOFTWARE - Програмное обеспечение сервера.

Пример:Apache/1.0

AUTH_TYPE, REMOTE_USER - Эти переменные определены в том случае,когда запрошенный ресурс требует аутентификации пользователя.

Переменные заголовка HTTP-запроса.

За исключением тех строк из заголовка HTTP-запроса, которые уже названы, сервер приделывает строкам префикс HTTP_ и заменяет знаки '-' на '_':

HTTP_ACCEPT - Давая запрос на сервер браузер обычно рассчитывает получить информацию определенного формата,и для этого он в заголовке запроса указывает поле Accept:,Отсюда скрипту поступает список тех MIME,которые браузер готов принять в качестве ответа от сервера.

Пример:HTTP_ACCEPT=text/html,text/plain,image/gif

HTTP_USER_AGENT - Браузер обычно посылает на сервер и информацию о себе,чтоб базируясь на знании особенностей и недостатков конкретных браузеров CGI-скрипт мог выдать информацию с учетом этого. Например,разные браузеры могут поддерживать или не поддерживать какие-то HTML тэги.

Пример:HTTP_USER_AGENT=Mozilla/2.01 Gold(Win95;I)

И другие всего их около 30.

CGI-скрипт получает доступ к значениям этих переменных через функции операционной системы (в разных операционных системах это реализуется по разному), тем самым CGI-скрипт получает исчерпывающую информацию об HTTP-запросе. А тело запроса (если оно конечно есть) поступает на STDIN (стандартный поток ввода) скрипта. Размером CONTENT_LENGTH байт.

Заголовок ответа CGI программы.

Теперь обсудим каким образом CGI-скрипт должен посылать информацию Web-серверу с точки зрения CGI.

Если CGI-скрипт хочет послать что-то в ответ, то он должен сделать это следующим образом. Все выводимые данные должны помещаться в STDOUT скрипта. При этом должен обязательно присутствовать CGI-заголовок (CGI-Header). В CGI-заголовке могут быть следующие поля:

Content-Type: – должно обязательно присутствовать, если есть тело ответа. Определяет MIME-тип ответа.

Например: Content-Type:text/html

Location: - Должно содержать URL – ресурса на который перенаправляется запрос, как правило в этом случае больше ничего не указывается (т.к запрос перенаправляется на другой сервер)

Например: Location:http://www.idsoftware.com/index.html

Status: - Содержит код завершения работы CGI-скрипта. Если не указан то подразумевается 200 Ok

Например: Status:404 Not found

Далее после CGI заголовок на STDOUT посылается пустая строка, которая отделяет заголовок от тела ответа. И после ее посылается собственно тело ответа тип которого был указан в Content-Type (рисунок, текст HTML или другое).

Web-сервер получив через STDOUT информацию поступившую от CGI-скрипта формирует на базе ее HTTP-ответкоторый и посылается клиенту (броузеру).

Лекция 6 Создание форм. Методы передачи параметров (GET, POST)

PHP-сценарии обработки HTML-форм

В основном для передаче параметров из форм используются методы POST и GET. Главное отличие методов POST и GET заключается в способе передачи информации. В методе GET параметры передаются через адресную строку (URL), т.е. в HTTP-заголовке запроса, в то время как в методе POST параметры передаются через тело HTTP-запроса и никак не отражаются в адресной строке.

Кнопки - Тег <BUTTON>

Тег <BUTTON> создает на веб-странице кнопки и по своему действию напоминает результат, получаемый с помощью тега <INPUT> (с параметром type="button | reset | submit"). В отличие от этого тега, <BUTTON> предлагает расширенные возможности по созданию кнопок. Например, на подобной кнопке можно размещать любые элементы HTML, в том числе изображения. Используя стили можно определить вид кнопки путем изменения шрифта, цвета фона, размеров и других параметров.

Теоретически, тег <BUTTON> должен располагаться внутри формы, устанавливаемой элементом <FORM>. Тем не менее, браузеры не выводят сообщение об ошибке и корректно работают с тегом <BUTTON>, если он встречается самостоятельно. Однако, если необходимо результат нажатия на кнопку отправить на сервер, помещать <BUTTON> в контейнер <FORM> обязательно. Закрывающий тег </BUTTON> обязателен.

Параметры:

- disabled - блокирует доступ и изменение элемента.
- type - тип кнопки
- value - Значение кнопки, которое будет отправлено на сервер или прочитано с помощью скриптов.

<button>Кнопка с текстом</button>

Параметр DISABLED Блокирует доступ и изменение кнопки. Она в таком случае отображается серой и недоступной для активации пользователем. Кроме того, такая кнопка не может получить фокус путем нажатия на клавишу Tab, мышью или другим способом. Тем не менее, такое состояние кнопки можно изменять через скрипты.

<form>

<p><button>Активная кнопка</button>

<button disabled>Неактивная кнопка</button></p>

</form>

Параметр TYPE Определяет тип кнопки, который устанавливает ее поведение в форме. По внешнему виду кнопки разного типа никак не различаются, но у каждой такой кнопки свои функции. Значение по умолчанию: button.

Аргументы:

button - Обычная кнопка.

reset - Кнопка для очистки введенных данных формы и возвращения значений в первоначальное состояние.

Submit - Кнопка для отправки данных формы на сервер.

```
<form method='post'>
```

```
<p><input type="text" name="user"></p>
```

```
<p><button type="reset">Очистить форму</button>
```

```
<button type="submit">Отправить форму</button></p>
```

```
</form>
```

```
<?php
```

```
echo $_POST['user']; // Если выбран хоть 1 элемент
```

```
?>
```

Параметр VALUE Определяет значение кнопки, которое будет отправлено на сервер. На сервер отправляется пара «имя=значение», где имя задается параметром name тега <BUTTON>, а значение — параметром value. Значение может, как совпадать с текстом на кнопке, так быть и самостоятельным. Также параметр value применяется для доступа к данным через скрипты.

```
<form action="/html/example/handler.php">
```

```
<p><input type="text" name="user"></p>
```

```
<p><button value="11111010101" name="hidden" type="submit">
```

```
Отправить форму</button></p>
```

```
</form>
```

Кнопка (input type=button)

```
<form>
```

```
<input type=button name="save" value="Привет!">
```

```
</form>
```

Кнопка с изображением (input type=image)

```
<button>
```

```

```

Кнопка с рисунком

```
</button></p>
```

Кнопки с изображениями аналогичны по действию кнопке Submit, но представляют собой рисунок. Для этого задаем type=image и src="image.gif".

```
<form>
```

```
<input type="image" src="image.gif" name="sub" />
```

```
</form>
```

Когда пользователь щелкнет где-нибудь на изображении, соответствующая форма будет передана на сервер с двумя дополнительными переменными - sub_x и sub_y. Они содержат координаты нажатия пользователя на изображение. Опытные программисты могут заметить, что на самом деле имена переменных, отправленных браузером, содержат точку, а не подчеркивание, но PHP автоматически конвертирует точку в подчеркивание.

Кнопка отправки формы (input type=submit)

Служит для отправки формы сценарию. При создании кнопки для отправки формы необходимо указать 2 атрибута: type="submit" и value="Текст кнопки". Атрибут name необходим, если кнопка не одна, а несколько и все они созданы для разных операций, например кнопки "Сохранить", "Удалить", "Редактировать" и т.д. После нажатия на кнопку сценарию передается строка имя=текст кнопки.

```
<input type="submit" name="Submit" value='Отправить'>
```

PHP-сценарий не требуется.

1.4. Массив кнопок (submit) для выбора варианта действий

```
<form method='post'>
  <input type=submit name="save" value="first">
  <input type=submit name="save" value="pref">
  <input type=submit name="save" value="next">
  <input type=submit name="save" value="last">
</form>
<?php
$action = $_POST['save'];
switch ($action)
{
  case 'first': echo 'Первый'; break;
  case 'pref' : echo 'Предыдущий'; break;
  case 'next' : echo 'Следующий'; break;
  case 'last' : echo 'Последний'; break;
}
?>
```

Кнопка сброса формы (Reset)

При нажатии на кнопку сброса (reset), все элементы формы будут установлены в то состояние, которое было задано в атрибутах по умолчанию, причем отправка формы не производится.

```
<input type="reset" name="Reset" value="Очистить форму">
```

PHP-сценарий не требуется.

Флажок (checkbox)

Флажки checkbox предлагают пользователю ряд вариантов, и разрешает произвольный выбор (ни одного, одного или нескольких из них).

```
<form method='post'>
  <input type='checkbox' name='chb[0]' value="white">Белый<br>
  <input type='checkbox' name='chb[1]' value="green">Зеленый<br>
```

```

<input type='checkbox' name='chb[2]' value="blue">Синий<br>
<input type='checkbox' name='chb[3]' value="red">Красный<br>
<input type='checkbox' name='chb[4]' value="black">Черный<br>
<input type='submit' name="Submit" value='Отправить'>
</form>

```

```

<?php

```

```

if (!empty($_POST['chb']))
{
    $chb = $_POST['chb'];
    foreach($chb as $index => $go)
    {
        echo $index." - > ".$go."<br>";
    };
};

```

Переключатель(radio**)**

Переключатели **radio** предлагают пользователю ряд вариантов, но разрешает выбрать только один из них.

Пример 1.

```

<form method='post'>
<input name="mycolor" type="radio" value="white">Белый<br>
<input name="mycolor" type="radio" value="green"
checked>Зеленый<br>
<input name="mycolor" type="radio" value="blue">Синий<br>
<input name="mycolor" type="radio" value="red">Красный<br>
<input name="mycolor" type="radio" value="black">Черный<br>
<input type='submit' name="Submit" value='Отправить'>
</form>

```

```

<?php echo $_POST['mycolor']; ?>

```

Пример 2.

```

<form method=GET>
// первый набор кнопок
<input type='radio' name='rdi[0]' value='1'>
<input type='radio' name='rdi[0]' value='2'>
<input type='radio' name='rdi[0]' value='3'><br>
// второй набор кнопок
<input type='radio' name='rdi[1]' value='1'>
<input type='radio' name='rdi[1]' value='2'>
<input type='radio' name='rdi[1]' value='3'><br>
// третий набор кнопок
<input type='radio' name='rdi[2]' value='1'>
<input type='radio' name='rdi[2]' value='2'>
<input type='radio' name='rdi[2]' value='3'><br>
<input type='submit' value='Отправить'>
</form>

```



```
<?php
$rdi = $_GET['rdi'];
while(list($key,$val) = @each($rdi))
    echo "ключ - $key, значение - $val<br>\n";
?>
```

Текстовое поле (text)

При создании обычного текстового поля размером size и максимальной допустимой длиныmaxlength символов, атрибут type принимает значение text. Если указан параметр value, то поле будет отображать указанный в переменной value. При создании поля не забывайте указывать имя поля, т.к. этот атрибут является обязательным.

```
<form method='post'>
    <input type="text" name="txtName" size="40" maxlength="35"
value="Текст по умолчанию">
    <input type='submit' name="Submit" value='Отправить'>
</form>
<?php echo $_POST['txtName']; ?>
```

Поле для ввода пароля (password)

Полностью аналогичен текстовому полю, за исключением того, что символы, набираемые пользователем, не будут отображаться на экране.

```
<form method='post'>
    <input type="password" name="txtName" size="40" maxlength="35">
    <input name="Submit" type='submit' value='Отправить'>
</form>
<?php echo $_POST['txtName']; ?>
```

Скрытое текстовое поле (hidden)

Позволяет передавать сценарию какую то служебную информацию, не отображая её на странице.

```
<form method='post'>
    <input type="hidden" name="txtName" value="Это скрытый текст">
    <input name="Submit" type='submit' value='Отправить'>
</form>
<?php echo $_POST['txtName']; ?>
```

Выпадающий список (select)

Тэг <select> представляет собой выпадающий или раскрытый список, при этом одновременно могут быть выбраны одна или несколько строк. Но будет передано значение последней выбранной кнопки.

Список начинается с парных тегов <select></select>. Теги <option></option> позволяют определить содержимое списка, а параметр value определяет значение строки. Если в теге <option>указан параметр selected, то строка будет изначально выбранной. Параметр size задает, сколько строк будет занимать список. Если size равен 1, то список будет выпадающим. Если указан атрибут multiple, то разрешено выбирать несколько элементов из списка. Но эта схема практически не используется, а при size = 1 не имеет смысла.

```

<form method='post'>
  <select name="color"> <!--выпадающий список-->
    <!--<select name="color" size=3> <!--список с прокруткой-->
    <option value="white">Белый</option>
    <option value="green">Зеленый</option>
    <option value="blue">Синий</option>
    <option value="red">Красный</option>
    <option value="black">Черный</option>
  </select>
  <input name="Submit" type='submit' value='Отправить'>
</form>
<?php
f (!empty($_POST['color'])) { echo $_POST['color']; };
?>

```

Если необходимо создать выпадающий с предсказуемой последовательностью. Например, список с годами с 2000 по 2050. То используется следующий прием.

```

<form method='post'>
  <select name=years>
<?php
$year = 2000;
for ($i = 0; $i <= 50; $i++) // Цикл от 0 до 50
{
  $new_years = $year + $i; // Формируем новое значение
  echo '<option value='.$new_years.'>'.$new_years.'</option>'; //строка
}
?>
</select>
<input name="Submit" type='submit' value='Отправить'>
</form>
<?php
if (!empty($_POST['years']))
{
  echo $_POST['years'];
};
?>

```

Многострочное поле ввода текста (textarea)

Многострочное поле ввода текста позволяет отправлять не одну строку, а сразу несколько. При необходимости можно указать атрибут `readonly`, который запрещает редактировать, удалять и изменять текст, т.е. текст будет предназначен только для чтения. Если необходимо чтобы текст был изначально отображен в многострочном поле ввода, то его необходимо поместить между тэгами `<textarea></textarea>`.

Существует параметр `wrap` – задание переноса строк. Возможные значения:

off - отключает перенос строк;
virtuals - показывает переносы строк, но отправляет текст как он введен;

physical - переносы строк оставляются в исходном виде.

По умолчанию тег <textarea> создает пустое поле шириной в 20 символов и состоящее из 2строк.

```
<form method='post'>
<textarea name="text" cols="40" rows="5">
Первоначально вставленный текст
</textarea>
<br>
<!--
<textarea name="text" cols="40" rows="5" readonly>
Первоначально вставленный текст
</textarea>
<br>
-->
<input name="Submit" type='submit' value='Отправить'>
</form>
<?php echo $_POST['text']; ?>
```

Для того, чтобы в многострочном текстовом поле соблюдалось html-форматирование (перенос строк по средством тега
 или <br \>), то используйте функцию nl2br():

```
<form method='post'>
<textarea name="text" cols="40" rows="5">
Первоначально вставленный строка 1
Первоначально вставленный строка 2
Первоначально вставленный строка 3
</textarea>
<input name="Submit" type='submit' value='Отправить'>
</form>
<?php echo nl2br($_POST['text']); ?>
```

Кнопка для загрузки файлов (browse)

Служит для реализации загрузки файлов на сервер. При создании текстового поля также необходимо указать тип поля type как "file".

```
<form method="post">
Загрузить файл:<br>
<input name="filename" type="file"><br>
<input type="submit" value="Отправить">
</form>
<?php echo ($_POST['filename']); ?
```

Способов, предоставляемых протоколом HTTP, немного. Это важная информация. Никаких других способов нет. На практике используются

два: GET - это когда данные передаются в адресной строке, например, когда пользователь жмет ссылку. POST - когда он нажимает кнопку в форме.

Метод GET

Чтобы передать данные методом GET не надо создавать на HTML-странице форму (использовать формы для запросов методом GET вам никто не запрещает - но это тупость) - достаточно ссылки на документ с добавлением строки запроса которая может выглядеть как переменная=значение пары объединяются с помощью амперсанда & а к URL страницы строка присоединяется с помощью вопросительного знака «?».

Но можно не использовать пары ключ=значение если надо передать всего одну переменную для этого надо после знака вопроса написать ЗНАЧЕНИЕ (не имя) переменной.

Преимущество передачи параметров таким способом заключается в том что клиенты которые не могут использовать метод POST (например, поисковые машины) все же смогут просто пройдя по ссылке передать параметры скрипту и получить содержимое.

Недостаток в том, что просто изменив параметры в адресной строке пользователь может повернуть ход сценария непредсказуемым образом, это создает огромную дыру в безопасности в сочетании с неопределенными переменными и register_globals on или кто-нибудь может узнать значение важной переменной (например ID-сессии) просто посмотрев на экран монитора.

Для чего следует использовать:

- для доступа к общедоступным страницам с передачей параметров (повышение функциональности)

- передача информации не влияющей на уровень безопасности

Для чего не следует использовать:

- для доступа к защищенным страницам с передачей параметров

- для передачи информации влияющей на уровень безопасности

- для передачи информации не подлежащей модифицированию пользователем (некоторые передают текст SQL-запросов).

Метод POST

Передать данные методом POST можно только с помощью формы на HTML странице. Основное отличие POST от GET в том, что данные передаются не в заголовке запроса а в теле, следовательно пользователь их не видит. Модифицировать может только изменив саму форму.

Преимущество:

- большая безопасность и функциональность запросов с помощью форм методом POST.

Недостаток:

- меньшая доступность.

Для чего следует использовать:

- для передачи большого объема информации (текст, файлы..);

- для передачи любой важной информации;

- для ограничения доступа (например, использовать для навигации только форму - возможность доступная не всем программам-роботам или грабберам-контента).

Для чего не следует использовать:

- нет ограничений.

Лекция 7 HTTP - cookie. Атрибуты cookie. Сессии. Идентификация пользователя.

PHP прозрачно поддерживает HTTP-куки. Куки это механизм хранения данных в удалённом браузере и отслеживания и идентифицирования пользователей с их помощью. Вы можете установить куки функцией `setcookie()`. Куки являются частью HTTP header'a, поэтому `setcookie()` обязана вызываться до отправления любого вывода браузеру. Это то же самое ограничение, что и для `header()`. Вы можете использовать функции буферизации вывода для задержки вывода скрипта до тех пор, пока вы не определите, устанавливать ли куки и отправлять ли какие-нибудь header'ы.

Любая кука, отправляемая вам с клиента, будет автоматически конвертирована в PHP-переменную точно так же, как GET и POST-данные, в зависимости от переменных конфигурации `register_globals` и `variables_order`. Если вы хотите присвоить несколько переменных одной куке, просто добавьте [] в имя куки.

В PHP 4.1.0 и позднее автоглобальный массив `$_COOKIE` всегда будет установлен любой кукой, отправленной клиентом. `$HTTP_COOKIE_VARS` также устанавливается в более ранних версиях PHP, когда установлена переменная конфигурации `track_vars`.

Дополнительные детали, включая замечания о багах браузера, см. в описании функции `setcookie()`.

Функция `setcookie`

(PHP 3, PHP 4)

`setcookie` - отправляет куки.

Описание

`int setcookie (string name [, string value [, int expire [, string path [, string domain [, int secure]]]])`

`setcookie()` определяет куку для отправки вместе с остальной header-информацией. Куки обязаны быть отправлены до любых других шапок/headers (это ограничение кук, а не PHP). Это требует, чтобы вы помещали вызовы этой функции перед тэгами `<html>` или `<head>`.

Все аргументы, кроме `name`, являются необязательными. Если имеется только аргумент `name`, кука с этим именем будет удалена с удалённого клиента. Вы можете также заместить любой аргумент пустой строкой (`""`), чтобы пропустить этот аргумент. Аргументы `expire` и `secure` это целые числа/integer и они не могут быть пропущены с помощью пустой строки. В них используйте нуль (0). Аргумент `expire` это обычное Unix time integer, возвращаемое функциями `time()` или `mktime()`. Аргумент `secure` указывает,

что данная кука должна передаваться только через секретное HTTPS-соединение.

После того как куки установлены, доступ к ним может быть получен при загрузке следующей страницы через массив `$_COOKIE` (который вызывается `$HTTP_COOKIE_VARS` в версиях PHP до 4.1.0).

Обычные ловушки:

Куки будут невидимы до тех пор, пока не будет загружена следующая страница.

Куки обязаны быть удалены с теми же параметрами, с которыми были установлены.

В PHP 3 множественные вызовы `setcookie()` в том же скрипте могут быть выполнены в реверсном порядке. Если вы пытаетесь удалить одну куку до вставки другой, вы должны сделать вставку до удаления. В PHP 4 множественные вызовы `setcookie()` выполняются в порядке вызова.

Далее идут примеры отправки кук:

Пример. Отправка кук функцией `setcookie()`

```
setcookie ("TestCookie", $value);
setcookie ("TestCookie", $value,time()+3600); /* период действия - 1
час */
setcookie ("TestCookie", $value,time()+3600, "/~rasmus/", ".utoronto.ca",
1);
```

При удалении куки вы должны убедиться, что дата окончания действия прошла, чтобы переключить механизм в вашем браузере. Далее идут примеры удаления куки, созданной в предыдущем примере:

Пример. Удаление куки с помощью `setcookie()`

```
// установить дату окончания действия на один час назад
setcookie ("TestCookie", "", time() - 3600);
setcookie ("TestCookie", "", time() - 3600, "/~rasmus/", ".utoronto.ca", 1);
```

Обратите внимание, что часть `value` куки будет автоматически `urlencoded` при отправке куки, и, когда она получена, она автоматически декодируется и присваивается переменной с тем же именем, что и имя куки. Для просмотра содержимого нашей тестовой куки в скрипте просто используйте один из следующих примеров:

```
echo $TestCookie;
echo $_COOKIE["TestCookie"];
```

Вы можете также установить куки массива, используя нотацию в имени куки. Это даёт эффект установки столько кук, сколько элементов в этом массиве, но, когда кука получается скриптом, значения помещаются в массив с именем куки:

```
setcookie ("cookie[three]", "cookiethree");
setcookie ("cookie[two]", "cookietwo");
setcookie ("cookie[one]", "cookieone");
```

Раздел 3 Программирование на стороне сервера на PHP

Лекция 8 Введение в PHP

Главная область применения PHP - написание скриптов, работающих на стороне сервера; таким образом, PHP способен выполнять все то, что выполняет любая другая программа CGI, например, обрабатывать данные форм, генерировать динамические страницы или отсылать и принимать cookies. Но PHP способен выполнять намного больше.

Существуют три основных области применения PHP.

- Создание скриптов для выполнения на стороне сервера. PHP традиционно и наиболее широко используется именно таким образом. Для этого вам будут необходимы три вещи. Интерпретатор PHP (в виде программы CGI или серверного модуля), веб-сервер и браузер. Для того чтобы можно было просматривать результаты выполнения PHP-скриптов в браузере, нужен работающий веб-сервер и установленный PHP. Просмотреть вывод PHP-программы можно в браузере, получив PHP-страницу, сгенерированную сервером. В случае, если вы просто экспериментируете, вы вполне можете использовать свой домашний компьютер вместо сервера. За более подробными сведениями обратитесь к главе Советы по установке.
- Создание скриптов для выполнения в командной строке. Вы можете создать PHP-скрипт, способный запускаться без сервера или браузера. Все, что вам потребуется - парсер PHP. Такой способ использования PHP идеально подходит для скриптов, которые должны выполняться регулярно, например, с помощью cron (на платформах *nix или Linux) или с помощью планировщика задач (Task Scheduler) на платформах Windows. Эти скрипты также могут быть использованы в задачах простой обработки текстов. За дополнительной информацией обращайтесь к главе Использование PHP в среде командной строки.
- Создание оконных приложений, выполняющихся на стороне клиента. Возможно, PHP является не самым лучшим языком для создания подобных приложений, но, если вы очень хорошо знаете PHP и хотели бы использовать некоторые его возможности в своих клиентских приложениях, вы можете использовать PHP-GTK для создания таких приложений. Подобным образом вы можете создавать и кросс-платформенные приложения. PHP-GTK является расширением PHP и не поставляется вместе с основным дистрибутивом PHP. Если вы заинтересованы, посетите » сайт PHP-GTK.

PHP доступен для большинства операционных систем, включая Linux, многие модификации Unix (такие как HP-UX, Solaris и OpenBSD), Microsoft Windows, Mac OS X, RISC OS, и многие другие. Также в PHP включена поддержка большинства современных веб-серверов, таких как Apache, IIS и многих других. В принципе, подойдет любой веб-сервер, способный использовать бинарный файл FastCGI PHP, например, lighttpd или nginx. PHP

может работать в качестве модуля или функционировать в качестве процессора CGI.

Таким образом, выбирая PHP, вы получаете свободу выбора операционной системы и веб-сервера. Более того, у вас появляется выбор между использованием процедурного или объектно-ориентированного программирования (ООП) или же их сочетания.

PHP способен генерировать не только HTML. Доступно формирование изображений, файлов PDF и даже роликов Flash (с использованием libswf и Ming), создаваемых «на лету». PHP также способен генерировать любые текстовые данные, такие, как XHTML и другие XML-файлы. PHP может осуществлять автоматическую генерацию таких файлов и сохранять их в файловой системе вашего сервера вместо того, чтобы отдавать клиенту, организуя, таким образом, серверный кэш для вашего динамического контента.

Одним из значительных преимуществ PHP является поддержка широкого круга баз данных. Создать скрипт, использующий базы данных, - невероятно просто. Можно воспользоваться расширением, специфичным для отдельной базы данных (таким как mysql) или использовать уровень абстракции от базы данных, такой как PDO, или подсоединиться к любой базе данных, поддерживающей Открытый Стандарт Соединения Баз Данных (ODBC), с помощью одноименного расширения ODBC. Для других баз данных, таких как CouchDB, можно воспользоваться cURL или сокетами.

PHP также поддерживает «общение» с другими сервисами через такие протоколы, как LDAP, IMAP, SNMP, NNTP, POP3, HTTP, COM (на платформах Windows) и многих других. Кроме того, вы получаете возможность работать с сетевыми сокетами напрямую. PHP поддерживает стандарт обмена сложными структурами данных WDDX практически между всеми языками веб-программирования. Обращая внимание на взаимодействие между различными языками, следует упомянуть о поддержке объектов Java и возможности их использования в качестве объектов PHP.

PHP имеет много возможностей по обработке текста, включая регулярные выражения Perl (PCRE) и много других расширений и инструментов для обработки и доступа к XML документам. В PHP обработка XML-документов стандартизирована и происходит на базе мощной библиотеки libxml2, расширив возможности обработки XML добавлением новых расширений SimpleXML, XMLReader и XMLWriter.

Есть еще многих других интересных расширений, которые можно просмотреть как в алфавитном порядке, так и по категориям. Есть еще много дополнительных PECL расширений, которые также могут (а может и нет) быть документированы в данном руководстве, такие как » XDebug.

Лекция 9 Основы синтаксиса

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

Perl-скрипт:

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

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

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

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

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

Вообще, универсальный (то есть - гарантированно работающий при любой конфигурации PHP), но более длинный способ спецификации PHP-кода - тэги `<?php ... ?>`. Такая длинная форма записи используется при совмещении XML и PHP, так как тэг `<? ... ?>` используется в стандарте XML. За распознавание тэга `<? как начала PHP-блока отвечает директива short_open_tag файла php.ini(по умолчанию - включена). Если вы хотите разрабатывать скрипты, работающие независимо от данной настройки, используйте длинный открывающий тэг <?php. Я буду использовать сокращенную форму.`

Рассмотрим простой пример.

```
<html>
<head><title>Hello World</title></head>
<body><h1>Hello World!</h1>
<p>Текущая дата:
<?
echo date("d.m.Y");
?>
</body>
</html>
```

Для выполнения примеров, скопируйте их в файл, расположенный в каталоге, соответствующий директиве DocumentRoot файла конфигурации Apache httpd.conf (например, в файл с именем test.php), и выполните их,

обратившись к сохраненному скрипту (test.php) из адресной строки браузера (http://localhost/test.php). Ну а если вы еще не установили Apache+PHP (как же так? ;), обратитесь к предыдущей главе.

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

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

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

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

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

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

Переменные и типы данных

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

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

Для присваивания переменной значения используется оператор =.

Пример:

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

```
?>
```

Данный код выведет: testtest100.

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

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

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

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

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

Рассмотрим еще один пример:

```
<?
```

```
$greeting = 'Привет';
```

```
$name = 'Вася';
```

```
$message = "$greeting, $name!";
```

```
echo $message;
```

```
?>
```

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

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

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

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

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

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

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

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

Немного изменим наш последний пример:

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

На этот раз мы не стали пользоваться "услужливостью" двойных кавычек: в строке 4 имена переменных и строковые константы записаны через оператор конкатенации (объединения строк). В PHP конкатенация

обозначается точкой - .. Результат выполнения этой программы аналогичен предыдущему примеру.

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

Имена переменных более четко визуально отделены от строковых значений, что лучше всего заметно в редакторе с подсветкой кода;

Интерпретатор PHP обрабатывает такую запись немного быстрее;

PHP сможет более четко отследить опечатку в имени переменной;

Вы не совершите ошибку, подобную следующей: `$message = "$greetingVasya"` - PHP в данном случае выведет не "ПриветVasya", а пустую строку, ибо `$greetingVasya` распознается как имя переменной, а таковой у нас нет.

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

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

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

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

Лекция 10 Управляющие конструкции

Условные операторы

`if`

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

`<?`

`$i = 10;`

`$j = 5 * 2;`

`if ($i == $j)`

```
echo 'Переменные $i и $j имеют одинаковые значения';  
else  
echo 'Переменные $i и $j имеют различные значения';  
?>
```

Здесь используется оператор `if..else` - условный оператор.

В общем виде он выглядит так:

```
if (условие)  
    выражение_1;  
else  
    выражение_2;
```

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

`!=` - не равно;
`<` - меньше;
`>` - больше;
`<=` - меньше или равно;
`>=` - больше или равно.

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

Не путайте оператор сравнения `==` с оператором присваивания `=`! Если вы допустите такую ошибку, условие всегда будет верным, если присваивается значение, соответствующее булевому `true`, и всегда ложным - если значение соответствует `false`. (См. выше о булевых значениях и чему они соответствуют).

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

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

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

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

```
<?  
$i = 10;  
$j = 11;
```

```

    if ($i > $j)
    $diff = $j - $i;
    echo '$j больше, чем $i; разность между $j и $i составляет ' . $diff;
//НЕВЕРНО!

```

?>

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

<?

```

    $i = 10;
    $j = 11;
    if ($i > $j) {
    $diff = $j - $i;
    echo '$j больше, чем $i; разность между $j и $i составляет ' . $diff;
    }

```

?>

Теперь все работает правильно.

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

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

<?

```

    $i = 10;
    $j = 11;
    if ($i > $j) {
    echo '$i больше, чем $j';
    } else if ($i < $j) {
    echo '$i меньше, чем $j';
    } else { // ничего, кроме равенства, не остается :)
    echo '$i равно $j';
    }

```

?>

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

Кстати, в строке 8 - комментарий. Это информация для человека, РНР ее игнорирует. Комментарии бывают двух видов: однострочный, как здесь - начинается с // и распространяется до конца строки, и многострочный -

комментарием считается все, что расположено между парами символов `/*` и `*/`.

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

```
switch
```

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

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

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

```
<?  
$i = 1;  
  
switch ($i) {  
case 1:  
echo 'один';  
break;  
case 2:  
echo 'два';  
break;  
case 3:  
echo 'три';  
break;  
default:  
echo 'я умею считать только до трех! ;)';  
}  
?>
```

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

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

Специальное условие `default` соответствует "всему остальному" (аналог `else` в `if...else if...else`). `default` всегда располагается последним, так что `break`

здесь необязателен. Как и в случае с else, условие default может отсутствовать.

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

```
<?
$i = 1;
switch ($i) {
case 0: // break отсутствует умышленно!
case 1:
echo 'ноль или один';
break;
case 2:
echo 'два';
break;
case 3:
echo 'три';
break;
}
?>
```

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

Циклы

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

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

Цикл while (строка 3) работает следующим образом. Сначала проверяется истинность выражения в скобках. Если оно не истинно, тело цикла (все, что расположено между последующими фигурными скобками - или, если их нет - следующая инструкция) не выполняется. Если же оно

истинно, после выполнения кода, находящегося в теле цикла, опять проверяется истинность выражения, и т.д.

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

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

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

`$i--`.

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

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

Запишем тот же пример в более краткой форме:

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

```
?>
```

И еще один вариант:

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

```
?>
```

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

`do..while`

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

Пример:

```
<?
$i = 1;
```

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

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

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

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

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

Перепишем "общий случай" цикла for в переложении на цикл while:

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

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

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

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

Операторы break и continue. Вложенные циклы

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

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

```

    if ($i == 5) break;
}
?>

```

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

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

```

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

```

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

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

```

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

```

Массивы

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

```

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

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

```

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

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

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

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

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

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

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

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

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

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

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

Вывод:

```

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

```

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

```

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

```

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

```

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

```

Вывод:

```

Array
(
    [1] => Assembler
    [2] => C++
    [3] => Pascal
    [scripting] => bash

```

```

[php] => PHP
[100] => Java
[101] => Perl
)

```

Цикл **foreach**

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

```

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

```

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

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

Конструкции **list** и **each**

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

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

'perl'. Соответственно, **list(\$a, \$b, \$c) = \$lang** дополнительно присвоит **\$c = 'basic'**.

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

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

```
<?
$browsers = array(
'MSIE' => 'Microsoft Internet Explorer 6.0',
'Gecko' => 'Mozilla Firefox 0.9',
'Opera' => 'Opera 7.50'
);
list($a, $b) = each($browsers);
list($c, $d) = each($browsers);
list($e, $f) = each($browsers);
echo $a.':'.$b."<br>\n";
echo $c.':'.$d."<br>\n";
echo $e.':'.$f."<br>\n";
?>
```

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

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

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

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

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

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

Константы

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

```
<?
define ('MY_NAME', 'Вася');
echo 'Меня зовут ' . MY_NAME;
?>
```

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

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

Лекция 11 Обработка запросов с помощью PHP

HTML-формы. Массивы \$_GET и \$_POST

Формы являются основным способом обмена данными между веб-сервером и браузером, то есть обеспечивают взаимодействие с пользователем - собственно, для чего и нужно веб-программирование.

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

Итак, возьмем уже знакомый вам по первой главе пример:

```
<html>
<body>
<?
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
echo '<h1>Привет, <b>' . $_POST['name'] . '</b></h1>!';
}
?>
<form method="POST" action="<?=$_SERVER['PHP_SELF']?>">
Введите Ваше имя: <input type="text" name="name">
<br>
<input type="submit" name="okbutton" value="OK">
</form>
</body>
</html>
```

Форма, приведенная в строках 8-12, содержит два элемента: name и okbutton. Атрибут method указывает метод отправки формы POST (см. главу 1), атрибут же action, указывающий URL, на который отправляется форма,

заполняется значением серверной переменной PHP_SELF - адресом выполняемого в данный момент скрипта.

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

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

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

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

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

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

```
<html>
<body>
<?
if (isset($_GET['name'])) {
echo '<h1>Привет, <b>' . $_GET['name'] . '</b></h1>!';
}
?>
<form action="<?=$_SERVER['PHP_SELF']?>">
Введите Ваше имя: <input type="text" name="name">
<br>
<input type="submit" value="ОК">
</form>
</body>
</html>
```

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

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

Кардинальное же отличие - в строке 4. Поскольку простой ввод адреса в строке браузера является GET-запросом, проверка **if**

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

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

Во многих старых книгах и статьях утверждается, что:

1. Данные как из GET, так и из POST-форм попадают непосредственно в переменные (в нашем случае - `$name`), причем POST-данные приоритетнее, т.е. "затирают" GET-данные;
2. Также данные GET и POST-форм хранятся соответственно в массивах `$HTTP_GET_VARS` и `$HTTP_POST_VARS`.

Эта информация давным-давно устарела. Уже года три как. Web-программирование, и, в частности, PHP развивается быстрыми темпами. Запомните этот момент, чтобы не попасть впросак со старыми книгами или скриптами.

Немного более сложный пример:

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

```
</html>
```

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

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

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

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

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

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

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

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

Введите Ваше имя:

Введите Ваш год рождения:

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

Разобрались? А теперь попробуйте найти ошибку в приведенном коде. Ну, не совсем ошибку, - но недочет.

htmlspecialchars()

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

Введите Ваше имя:

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

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

```
<html>
<body>
<?
$name = isset($_POST['name']) ? htmlspecialchars($_POST['name']) : "";
$year = isset($_POST['year']) ? htmlspecialchars($_POST['year']) : "";
if (isset($_POST['name'], $_POST['year'])) {
if ($_POST['name'] == "") {
echo 'Укажите имя!<br>';
} else if ($_POST['year'] < 1900 || $_POST['year'] > 2004) {
echo 'Укажите год рождения! Допустимый диапазон значений:
1900..2004<br>';
} else {
echo 'Здравствуй, ' . $name . '!<br>';
$age = 2004 - $_POST['year'];
echo 'Вам ' . $age . ' лет<br>';
}
echo '<hr>';
}
?>
```

```

        <form method="post" action="<?=$_SERVER['PHP_SELF']?>">
        Введите Ваше имя: <input type="text" name="name"
value="<?=$name?>">
        <br>
        Введите Ваш год рождения: <input type="text" name="year"
value="<?=$year?>">
        <input type="submit" value="OK">
    </form>
</body>
</html>

```

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

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

phpinfo()

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

Почему я упоминаю о ней в главе, посвященной формам? phpinfo() - удобное средство отладки. phpinfo(), помимо прочего, выводит значения всех \$_GET, \$_POST и \$_SERVER - переменных. Так что, если переменная формы "потерялась", самый простой способ обнаружить, в чем дело - воспользоваться функцией phpinfo. Для того, чтобы функция выводила только значения переменных (и вам не пришлось прокручивать десяток страниц), ее следует вызвать следующим образом: **phpinfo(INFO_VARIABLES);**, или - что абсолютно то же самое - **phpinfo(32);**.

Пример:

```

<html>
<body>
<form method="post" action="<?=$_SERVER['PHP_SELF']?>">
Введите Ваше имя: <input type="text" name="name">
<input type="submit" value="OK">
</form>
<?
phpinfo(32);
?>
</body>
</html>

```

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

Лекция 12 Функции в PHP

Функция может быть определена с использованием такого синтаксиса:

```
function foo ($arg_1, $arg_2, ..., $arg_n)
{
    echo "Пример \n";
    return $retval;
}
```

Внутри функции может появляться любой правильный код PHP, даже другие функции и определения классов.

В PHP 3 функции обязаны быть определены до обращения к ним. Такого требования нет в PHP 4.

PHP не поддерживает перегрузку/overloading функций; также невозможно разопределить или переопределить ранее объявленную функцию.

PHP 3 не поддерживает переменное количество аргументов функции, хотя аргументы по умолчанию поддерживаются (см. Значения аргументов по умолчанию). PHP 4 поддерживает и то, и другое: см. Списки аргументов переменного размера и статьи о функциях `func_num_args()`, `func_get_arg()` и `func_get_args()`.

Аргументы функции

Информация может передаваться в функцию через список аргументов, который является списком разделённых запятыми переменных и/или констант.

PHP поддерживает разбор аргументов по значению (по умолчанию), разбор по ссылке и значения по умолчанию. Списки аргументов Variable-length поддерживаются только в PHP 4 и позднее; см. Списки аргументов переменного размера и статью о функциях `func_num_args()`, `func_get_arg()` и `func_get_args()`. Аналогичный эффект может быть достигнут в PHP 3 путём передачи функции массива аргументов:

```
function takes_array($input)
{
    echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
```

Создание аргументов, передаваемых по ссылке

По умолчанию аргументы передаются в функцию по значению (поэтому, если вы изменяете значение аргумента внутри функции, оно не изменяется за пределами этой функции). Если вы дать функции возможность модифицировать свои аргументы, вы обязаны передавать их по ссылке.

Если всегда хотите передавать аргументы по ссылке, вы можете ввести префикс-амперсанд (&) в имени аргумента в определении

функции:

```
function add_some_extra(&$string)
{
    $string .= 'and something extra.';
}
$str = 'This is a string, ';
add_some_extra($str);
echo $str;// выводит 'This is a string, and something extra.'
```

Значения по умолчанию аргументов

Функция может определить значения по умолчанию в стиле C++ для скалярных аргументов:

```
function makecoffee ($type = "cappuccino")
{
    return "Making a cup of $type.\n";
}
echo makecoffee ();
echo makecoffee ("espresso");
```

Вывод будет таким:

```
Making a cup of cappuccino.
Making a cup of espresso.
```

Значение по умолчанию обязано быть константным выражением, а не (например) переменной или членом класса.

Обратите внимание, что при использовании аргументов по умолчанию любые значения по умолчанию должны находиться справа от любых значений не по умолчанию; иначе что-нибудь может работать не так, как ожидалось. Рассмотрим следующий фрагмент кода:

```
function makeyogurt ($type = "acidophilus", $flavour)
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt ("raspberry"); // не будет работать так, как
ожидается
```

Вывод этого примера:

```
Warning!: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/php3test/functest.html on line 41
Making a bowl of raspberry .
```

Теперь сравним с этим:


```
function makeyogurt ($flavour, $type = "acidophilus")
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt ("raspberry"); // работает как положено
```

На выводе этого примера:

```
Making a bowl of acidophilus raspberry.
```

Списки аргументов переменного размера

В PHP 4 имеется поддержка списков аргументов переменного размера/variable-length в пользовательских функциях. Это довольно легко делается функциями `func_num_args()`, `func_get_arg()` и `func_get_args()`.

Специального синтаксиса не нужно, и списки аргументов могут по-прежнему явно предоставляться в определениях функций и будут вести себя как обычно.

Возвращаемые значения

Значения из функций возвращаются с помощью необязательного оператора `return`. Может быть возвращён любой тип, в том числе список и объект. Этот оператор немедленно останавливает выполнение функции и передаёт управление обратно на строчку, с которой функция была вызвана. Дополнительно см. `return()`.

```
function square ($num)
{
    return $num * $num;
}

echo square (4); // выводит '16'
```

Вы можете вернуть из функции несколько значений, но сходные результаты можно получить путём возвращения списка.

```
function small_numbers()
{
    return array (0, 1, 2);
}

list ($zero, $one, $two) = small_numbers();
```

Чтобы вернуть из функции ссылку, вы должны использовать операцию ссылки `&` в объявлении функции и в присвоении возвращаемого значения переменной:

```
function &returns_reference()
{
    return $someref;
}

$newref =& returns_reference();
```

Лекция 13 Объекты и классы в PHP

Класс это коллекция переменных и функций, работающих с этими переменными. Класс определяется с использованием следующего синтаксиса:

```
<?php
class Cart
{
    var $items; //Элементы в нашей shopping cart

    // Добавить $num артикулов $artnr в cart

    function add_item ($artnr, $num)
    {
        $this->items[$artnr] += $num;
    }

    // Изъять $num артикулов $artnr из cart

    function remove_item ($artnr, $num)
    {
        if ($this->items[$artnr] > $num) {
            $this->items[$artnr] -= $num;
            return true;
        } else {
            return false;
        }
    }
}
?>
```

Здесь определён класс Cart, состоящий из ассоциативного массива артикулов/articles в карте/cart и двух функций для добавления и удаления элементов этой cart.

Примечание: В PHP 4 только константные инициализаторы для var-переменных допустимы. Для инициализации переменных с неконстантными значениями вам необходима функция инициализации, которая вызывается

автоматически, когда конструируется из класса. такая функция называется конструктором (см. далее).

```
<?php
/* Это не будет работать в PHP 4. */
class Cart
{
    var $todays_date = date("Y-m-d");
    var $name = $firstname;
    var $owner = 'Fred ' . 'Jones';
    var $items = array("VCR", "TV");
}

/* Вот как это должно делаться. */
class Cart
{
    var $todays_date;
    var $name;
    var $owner;
    var $items;

    function Cart()
    {
        $this->todays_date = date("Y-m-d");
        $this->name = $GLOBALS['firstname'];
        /* etc. . . */
    }
}
?>
```

Классы это типы, то есть они являются шаблонами реальных переменных. Вы должны создавать переменную нужного типа операцией new.

```
<?php
$cart = new Cart;
$cart->add_item("10", 1);

$another_cart = new Cart;
$another_cart->add_item("0815", 3);
```

Здесь создаются объекты \$cart и \$another_cart, оба от класса Cart. Функция add_item() объекта \$cart вызывается для добавления 1 элемента артикула номер 10 в \$cart. 3 элемента артикула номер 0815 добавляются в \$another_cart.

И \$cart, и \$another_cart имеют функции add_item(), remove_item() и элементы переменных. Это разные функции и переменные. Вы можете

представить эти объекты как директории файловой системы. В файловой системе вы можете два разных файла README.TXT, если они находятся в разных директориях. Как и с директориями, где необходимо вводить полный путь к файлу, чтобы достичь его из директории верхнего уровня, вы должны специфицировать полное имя функции, которую хотите вызвать: В терминологии PHP, директория верхнего уровня это глобальное пространство имён/global namespace, а разделителем имён служит ->. Таким образом, имена \$cart->items и \$another_cart->items именуют две различные переменные. Заметьте, что переменная именуется \$cart->items, а не \$cart->\$items, то есть имя переменной в PHP имеет только один знак dollar.

```
// корректно, один символ $
$cart->items = array("10" => 1);

// неверно, поскольку $cart->$items становится $cart->""
$cart->$items = array("10" => 1);

// корректно, но может и не быть тем, что вы предполагаете:
// $cart->$myvar becomes $cart->items
$myvar = 'items';
$cart->$myvar = array("10" => 1);
```

Внутри определения класса вы не знаете, под каким именем объект будет доступен в вашей программе: на момент написания класса Cart не было известно, что объект будет называться \$cart или \$another_cart. Таким образом, вы не можете написать \$cart->items в самом классе Cart. Вместо этого, чтобы иметь возможность доступа к переменным и функциям внутри класса, можно использовать псевдопеременную \$this, которая может читаться как 'моя собственная' или 'текущий объект'. То есть '\$this->items[\$artnr] += \$num' можно прочесть как 'добавить \$num к счётчику \$artnr элемента моего собственного массива' или 'добавить \$num к счётчику \$artnr элемента массива внутри текущего объекта'.

Примечание: Имеются отличные функции для работы с классами и объектами. Вы можете просмотреть их в разделе Class/Object-функции.

Лекция 14 Работа с массивами данных. Работа со строками

Массивы

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

```
<?
$i = 1024;
```

```

$a[1] = 'abc';
$a[2] = 100;
$a['test'] = $i - $a[2];

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

```

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

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

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

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

```

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

```

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

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

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

```

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

```

Если явно не указывать индексы, то здесь проявляется свойство массивов PHP, характерное для числовых массивов в других языках:

очередной элемент будет иметь порядковый числовой индекс. Нумерация начинается с нуля. Пример:

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

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

Вывод:

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

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

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

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

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

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

```
$languages[] = 'Perl';
```

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

```
?>
```

Вывод:

Array

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

Синтаксис строк

Строковой литерал может специфицироваться тремя способами.

1. одинарными кавычками,
2. двойными кавычками,
3. heredoc-синтаксисом.

Одинарные кавычки

Простейший способ специфицировать строку - заключить её в одинарные кавычки (символ `'`).

Для специфицирования литеральной одинарной кавычки вам нужно мнемонизировать/escape её с помощью backslash (`\`), как во многих других языках. Если backslash должен появиться перед одинарной кавычкой или в конце строки, нужно его удвоить. Обратите внимание, что если вы захотите мнемонизировать любой другой символ, backslash также будет напечатан! Поэтому обычно нет необходимости мнемонизировать сам backslash.

Примечание: в PHP 3 при этом будет выведено предупреждение уровня E_NOTICE.

Примечание: в отличие от двух других видов синтаксиса, переменные здесь не разворачиваются (не обсчитываются), когда находятся внутри строки, заключённой в одинарные кавычки.

```
echo "Это простая строка.";
echo 'Вы можете также внедрять в строки символы newline,  
как здесь.';
echo 'Arnold once said: "I\'ll be back"';
// на выходе: ... "I'll be back"
echo 'Are you sure you want to delete C:\\*..*?';
```

```
// на выходе: ... delete C:\*.*?  
echo 'Are you sure you want to delete C:\*.*?';  
// на выходе: ... delete C:\*.*?  
echo 'I am trying to include at this point: \n a newline';  
// на выходе: ... this point: \n a newline
```

Двойные кавычки

Если строка заключена в двойные кавычки ("), РНР понимает большее количество escape-последовательностей (мнемоник) специальных символов:

Ещё раз напоминает, что если вы захотите мнемонизировать любой другой символ, backslash также будет напечатан!

Но самым важным свойством строки, заключённой в двойные кавычки, является то, что имена переменных разворачиваются.

Heredoc

Другой способ ограничения строки - использовать синтаксис heredoc ("<<<"). После <<< необходимо предоставить идентификатор, затем строку, а затем - тот же идентификатор как закрывающую кавычку.

Закрывающий идентификатор обязан начинаться в первом столбце строчки. Используемый идентификатор также обязан следовать тем же правилам именования, что и все другие метки в РНР: он может содержать только алфавитные символы, числа и символ подчёркивания и обязан начинаться с не-цифры или с символа подчёркивания.

Предупреждение!

Важно отметить, что строчка с закрывающим идентификатором не содержит больше никаких символов, за исключением, возможно, точки с запятой (;). Это означает, что идентификатор не может вводиться с отступом и что не может быть никаких пробельных символов и знаков табуляции до и после точки с запятой.

Самое, возможно, неприятное, что в конце строки не может быть также и символа carriage return (\r), только form feed, АКА newline (\n). Поскольку Microsoft Windows использует последовательность \r\n как терминатор строки, ваш heredoc может не сработать, если вы запишете ваш скрипт в редакторе под Windows. Однако большинство программ-редакторов дают возможность сохранять ваши файлы с терминатором строк UNIX.

Heredoc текст ведёт себя так же, как строка в двойных кавычках. Это значит, что вам не нужно мнемонизировать кавычки в heredocs, но можно продолжать использовать коды-мнемоники, перечисленные выше. Переменные разворачиваются, но с комплексными переменными в heredoc нужно работать так же внимательно, как и со строками.

Пример Heredoc-пример строк в кавычках

```
<?php
```



```

$str = <<<EOD
Пример строки,
захватывающей несколько строчек,
с использованием синтаксиса heredoc.
EOD;

/* Более сложный пример, с переменными. */
class foo
{
    var $foo;
    var $bar;

    function foo()
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';

echo <<<EOT
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should print a capital 'A': \x41
EOT;
?>

```

Примечание: поддержка heredoc была введена в PHP 4.

Разбор переменных

Когда строка специфицируется двойными кавычками или heredoc, переменные внутри неё разбираются.

Есть два типа синтаксиса: простой и сложный. Простой синтаксис более распространён и удобнее, он предоставляет способ разбора переменной, переменной массива или свойства объекта.

Сложный синтаксис был введён в PHP 4 и может распознаваться по фигурным скобкам { } вокруг выражения.

Простой синтаксис

Если обнаружен знак dollar (\$), разборщик захватывает как можно больше лексем для образования правильного имени. Заклучайте имя переменной в фигурные скобки, если вы хотите явным образом специфицировать конец имени.

```
$beer = 'Heineken';
echo "$beer's taste is great"; // не будет работать, "" это неправильный
echo "He drunk some $beers"; // работает, 's' это правильный символ
                        // для имён переменных
echo "He drunk some ${beer}s"; // работает
```

Аналогично вы можете заставить разобрать индекс массива или свойство объекта. В индексах массивов закрывающая квадратная скобка (]) обозначает конец индекса. Для свойств объекта применяются те же правила, что и для простых переменных, хотя со свойствами объекта невозможен трюк, как с переменными.

```
$fruits = array( 'strawberry' => 'red' , 'banana' => 'yellow' );
// заметьте, что это работает по-другому вне заковыченной строки.
echo "A banana is $fruits[banana].";
echo "This square is $square->width meters broad.";
// не будет работать. Для решения см. сложный синтаксис.
echo "This square is $square->width00 centimeters broad.";
```

Для чего-либо более сложного вы должны использовать сложный синтаксис.

Сложный (фигурный) синтаксис

Он называется сложным не потому, что сложен, а потому что вы можете включать таким способом сложные выражения.

Фактически вы можете включать любое значение, которое находится в пространстве имён строки с этим синтаксисом. Вы просто записываете выражение тем же способом, что и вне строки, а затем окружаете его символами { и }. Поскольку вы не можете заменить '{' мнемоникой, этот синтаксис будет распознаваться, только когда \$ идёт непосредственно после {.

(Используйте "{\\$" или "\{" для получения литерала "{\$"). Вот некоторые примеры для пояснения:

```
great = 'fantastic';
echo "This is { $great}"; // не будет работать, выведет: This is { fantastic }
echo "This is {$great}"; // работает, выведет: This is fantastic
echo "Это квадрат шириной {$square->width}00 сантиметров.";
echo "Это работает: {$arr[4][3]}";
// Неверно по той причине,
// что $foo[bar] неверно вне строки.
echo "Это неправильно: {$arr[foo][3]}";
echo "Вы должны сделать это так: {$arr['foo'][3]}";
echo "Вы можете даже записать {$obj->values[3]->name}";
echo "Это значение переменной $name: {${$name}}";
```

Доступ к символу в строке

Можно получить доступ к символам в строке путём специфицирования смещения с базой 0 в фигурных скобках для нужного символа.

Примечание: для обеспечения обратной совместимости вы можете по-прежнему использовать скобки массива. Однако этот синтаксис не рекомендуется, начиная с PHP 4.

Пример Некоторые примеры строк

```
<?php
/* Присвоение строкового значения. */
$str = "This is a string";

/* Присоединение к ней. */
$str = $str . " with some more text";

/* Другой способ присоединения с мнемоникой для newline. */
$str .= " and a newline at the end.\n";

/* Эта строка будет выглядеть '<p>Number: 9</p>' */
$num = 9;
$str = "<p>Number: $num</p>";

/* А эта - '<p>Number: $num</p>' */
$num = 9;
$str = '<p>Number: $num</p>';

/* Получить первый символ строки */
$str = "This is a test.";
$first = $str{0};

/* Получить последний символ строки */
$str = "This is still a test.";
$last = $str{strlen($str)-1};
?>
```

Используемые функции

Строки можно объединять (конкатенировать) с помощью операции '.' (точка). Заметьте, что операция сложения '+' здесь не работает.

Конвертация строк

Когда строка вычисляется как числовое значение, результирующее значение и его тип определяются так.

Строка вычисляется как float, если содержит любой из символов '.', 'e' или 'E'. Иначе она вычисляется как integer.

Значение даётся по начальной части строки. Если строка начинается с правильного числового значения, используется это значение. Иначе значение будет 0 (нуль). Верным числовым символом является знак с последующими

одной или более цифрами (с возможной десятичной точкой), с последующей необязательной экспонентой. Экспонента это 'e' или 'E' с последующими одной или более цифрами.

```
$foo = 1 + "10.5";           // $foo это float (11.5)
$foo = 1 + "-1.3e3";          // $foo это float (-1299)
$foo = 1 + "bob-1.3e3";       // $foo это integer (1)
$foo = 1 + "bob3";            // $foo это integer (1)
$foo = 1 + "10 Small Pigs";    // $foo это integer (11)
$foo = 4 + "10.2 Little Piggies"; // $foo это float (14.2)
$foo = "10.0 pigs " + 1;       // $foo это float (11)
$foo = "10.0 pigs " + 1.0;     // $foo это float (11)
```

Если вы хотите протестировать любой из примеров этого раздела, вы можете скопировать его и вставить следующую строку, чтобы посмотреть, что будет:

```
echo "\$foo==\$foo; type is " . gettype($foo) . "<br />\n";
```

Лекция 15 Работа с файловой системой

Данная лекция посвящена одному из важнейших аспектов PHP -- средствам файлового ввода/вывода. Как нетрудно предположить, входные и выходные потоки данных интенсивно используются при разработке web-приложений. Не ограничиваясь простым чтением/записью файлов, PHP предоставляет в распоряжение программиста средства просмотра и модификации серверной информации, а также запуска внешних программ. Этим средствам и посвящена настоящая лекция.

Проверка существования и размера файла

Прежде чем пытаться работать с файлом, желательно убедиться в том, что он существует. Для решения этой задачи обычно используются две функции:

`file_exists()` и `is_file()`.

`file_exists()`

Функция `file_exists()` проверяет, существует ли заданный файл. Если файл существует, функция возвращает `TRUE`, в противном случае возвращается `FALSE`. Синтаксис функции `file_exists()`:

`bool file_exists(строка файл)`

Пример проверки существования файла:

```
if (! file_exists($filename)) :
print "File $filename does not exist!";
endif:
is_file( )
```

Функция `is_file()` проверяет существование заданного файла и возможность выполнения с ним операций чтения/записи. В сущности, `is_file()` представляет собой более надежную версию `file_exists()`, которая проверяет не только факт существования файла, но и то, поддерживает ли он чтение и запись данных:

```
bool is_file(string файл)
```

Следующий пример показывает, как убедиться в существовании файла и возможности выполнения операций с ним:

```
$file = "somefile.txt";  
if (is_file($file)) :  
    print "The file $file is valid and exists!";  
else :  
    print "The file $file does not exist or it is not a valid file!";  
endif:
```

Убедившись в том, что нужный файл существует и с ним можно выполнять различные операции чтения/записи, можно переходить к следующему шагу -- открытию файла.

```
filesize( )
```

Функция `filesize()` возвращает размер (в байтах) файла с заданным именем или `FALSE` в случае ошибки. Синтаксис функции `filesize()`:

```
int filesize(string имя_файла)
```

Предположим, вы хотите определить размер файла `pastry.txt`. Для получения нужной информации можно воспользоваться функцией `filesize()`:

```
$fs = filesize("pastry.txt"); print "Pastry.txt is $fs bytes.";
```

Выводится следующий результат:

```
Pastry.txt is 179 bytes.
```

Прежде чем выполнять операции с файлом, необходимо открыть его и связать с файловым манипулятором, а после завершения работы с файлом его следует закрыть. Эти темы рассматриваются в следующем разделе.

Открытие и закрытие файлов

Прежде чем выполнять операции ввода/вывода с файлом, необходимо открыть его функцией `fopen()`.

```
fopen( )
```

Функция `fopen()` открывает файл (если он существует) и возвращает целое число -- так называемый файловый манипулятор (`file handle`). Синтаксис функции `fopen()`:

```
int fopen (string файл, string режим [, int включение_пути])
```

Открываемый файл может находиться в локальной файловой системе, существовать в виде стандартного потока ввода/вывода или представлять файл в удаленной системе, принимаемой средствами HTTP или FTP.

Параметр файл может задаваться в нескольких формах, перечисленных ниже:

Если параметр содержит имя локального файла, функция `fopen()` открывает этот файл и возвращает манипулятор.

Если параметр задан в виде `php://stdin`, `php://stdout` или `php://stderr`, открывается соответствующий стандартный поток ввода/вывода.

Если параметр начинается с префикса `http://`, функция открывает подключение HTTP к серверу и возвращает манипулятор для указанного файла.

Если параметр начинается с префикса `ftp://`, функция открывает подключение FTP к серверу и возвращает манипулятор для указанного файла. В этом случае следует обратить особое внимание на два обстоятельства: если сервер не поддерживает пассивный режим FTP, вызов `fopen()` завершается неудачей. Более того, FTP-файлы открываются либо для чтения, либо для записи.

При работе в пассивном режиме сервер ЯР ожидает подключения со стороны клиентов. При работе в активном режиме сервер сам устанавливает соединение с клиентом. По умолчанию обычно используется активный режим.

Если необязательный третий параметр `включение_пути` равен 1, то путь к файлу определяется по отношению к каталогу включаемых файлов, указанному в файле `php.ini`

Ниже приведен пример открытия файла функцией `fopen()`. Вызов `die()`, используемый в сочетании с `fopen()`, обеспечивает вывод сообщения об ошибке в том случае, если открыть файл не удастся:

```
$file = "userdata.txt"; // Некоторый файл
$fh = fopen($file, "a+") or die("File ($file) does not exist!");
```

Следующий фрагмент открывает подключение к сайту PHP (`http://www.php.net`):

```
$site = "http://www.php.net": // Сервер, доступный через HTTP
$sh = fopen($site., "r"); //Связать манипулятор с индексной страницей
```

Php.net

После завершения работы файл всегда следует закрывать функцией `fclose()`.

```
fclose ( )
```

Функция `fclose()` закрывает файл с заданным манипулятором. При успешном закрытии возвращается `TRUE`, при неудаче -- `FALSE`. Синтаксис функции `fclose()`:

```
int fclose(int манипулятор)
```

Функция `fclose()` успешно закрывает только те файлы, которые были ранее открыты функциями `fopen()` или `fsockopen()`. Пример закрытия файла:

```
$file = "userdata.txt";
if (file_exists($file)) :
    $fh = fopen($file, "r");
    // Выполнить операции с файлом
    fclose($fh);
else :
    print "File $file does not exist!";
endif;
```

Запись в файл

С открытыми файлами выполняются две основные операции -- чтение и запись.

`is_writable()`

Функция `is_writable()` позволяет убедиться в том, что файл существует и для него разрешена операция записи. Возможность записи проверяется как для файла, так и для каталога. Синтаксис функции `is_writable()`:

`bool is_writable (string файл)`

Одно важное обстоятельство: скорее всего, PHP будет работать под идентификатором пользователя, используемым web-сервером (как правило, «nobody»).

Запись в файл

Функция `fwrite()` записывает содержимое строковой переменной в файл, заданный файловым манипулятором. Синтаксис функции `fwrite()`:

`int fwrite(int манипулятор, string переменная [, int длина])`

Если при вызове функции передается необязательный параметр длина, запись останавливается либо после записи указанного количества символов, либо при достижении конца строки. Проверка возможности записи в файл продемонстрирована в следующем примере:

```
<?
// Информация о трафике на пользовательском сайте
$data = "08:13:00|12:37:12|208.247.106.187|Win98";
$filename = "somefile.txt";
// Если файл существует и в него возможна запись
if ( is_writable($filename) ) :
// Открыть файл и установить указатель текущей позиции в конец
файла
$fh = fopen($filename, "a+");
// Записать содержимое $data в файл
$ success - fwrite($fh, $data);
// Закрыть файл
fclose($fh); else :
print "Could not open $filename for writing";
endif;
?>
```

Функция `fputs()` является псевдонимом `fwrite()` и может использоваться всюду, где используется `fwrite()`.

`fputs()`

Функция `fputs()` является псевдонимом `fwrite()` и имеет точно такой же синтаксис. Синтаксис функции `fputs()`:

`int fputs(int манипулятор, string переменная [, int длина])`

Лично я предпочитаю использовать `fputs()`. Следует помнить, что это всего лишь вопрос стиля, никак не связанный с какими-либо различиями между двумя функциями.

Чтение из файла

Несомненно, чтение является самой главной операцией, выполняемой с файлами. Ниже описаны некоторые функции, повышающие эффективность чтения из файла. Синтаксис этих функций практически точно копирует синтаксис аналогичных функций записи.

`is_readable()`

Функция `is_readable()` позволяет убедиться в том, что файл существует и для него разрешена операция чтения. Возможность чтения проверяется как для файла, так и для каталога. Синтаксис функции `is_readable()`:

`bool is_readable (string файл)`

Скорее всего, PHP будет работать под идентификатором пользователя, используемым web-сервером (как правило, «nobody»), поэтому для того чтобы функция `is_readable()` возвращала `TRUE`, чтение из файла должно быть разрешено всем желающим. Следующий пример показывает, как убедиться в том, что файл существует и доступен для чтения:

```
if ( is_readable($filename) ) :
```

```
// Открыть файл и установить указатель текущей позиции в конец файла
```

```
$fh = fopen($filename, "r");
```

```
else :
```

```
print "$filename is not readable!";
```

```
endif;
```

```
fread( )
```

Функция `fread()` читает из файла, заданного файловым манипулятором, заданное количество байт. Синтаксис функции `fwrite()`:

```
int fread(int манипулятор, int длина)
```

Манипулятор должен ссылаться на открытый файл, доступный для чтения (см. описание функции `is_readable()`). Чтение прекращается после прочтения заданного количества байт или при достижении конца файла. Рассмотрим текстовый файл `pastry.txt`, приведенный в листинге 7.1. Чтение и вывод этого файла в браузере осуществляется следующим фрагментом:

```
$fh = fopen('pastry.txt', "r") or die("Can't open file!");
```

```
$file = fread($fh, filesize($fh));
```

```
print $file;
```

```
fclose($fh);
```

Используя функцию `filesize()` для определения размера `pastry.txt` в байтах, вы гарантируете, что функция `fread()` прочтает все содержимое файла.

Листинг 7.1. Текстовый файл `pastry.txt`

Recipe: Pastry Dough

1 1/4 cups all-purpose flour

3/4 stick (6 tablespoons) unsalted butter, chopped

2 tablespoons vegetable shortening 1/4 teaspoon salt

3 tablespoons water

`fgetc()`

Функция `fgetc()` возвращает строку, содержащую один символ из файла в текущей позиции указателя, или `FALSE` при достижении конца файла.

Синтаксис функции `fgetc()`:

`string fgetc (int манипулятор)`

Манипулятор должен ссылаться на открытый файл, доступный для чтения (см. описание функции `is_readable()` ранее в этой главе). В следующем примере продемонстрированы посимвольное чтение и вывод файла с использованием функции `fgetc()`:

```
$fh = fopen("pastry.txt", "r"); while (! feof($fh)) :
```

```
$char = fgetc($fh):
```

```
print $char; endwhile;
```

```
fclose($fh);
```

`fgets()`

Функция `fgets()` возвращает строку, прочитанную от текущей позиции указателя в файле, определяемом файловым манипулятором. Файловый указатель должен ссылаться на открытый файл, доступный для чтения (см. описание функции `is_readable()` ранее в этой главе). Синтаксис функции `fgets()`:

`string fgets (int манипулятор, int длина)`

Чтение прекращается при выполнении одного из следующих условий:

из файла прочитано длина -- 1 байт;

из файла прочитан символ новой строки (включается в возвращаемую строку);

из файла прочитан признак конца файла (EOF).

Если вы хотите организовать построчное чтение файла, передайте во втором параметре значение, заведомо превышающее количество байт в строке.

Пример построчного чтения и вывода файла:

```
$fh = fopen("pastry.txt", "r");
```

```
while (! feof($fh));
```

```
$line = fgets($fh, 4096);
```

```
print $line. "<br>";
```

```
endwhile;
```

```
fclose($fh);
```

`fgetss()`

Функция `fgetss()` полностью аналогична `fgets()` за одним исключением -- она пытается удалять из прочитанного текста все теги HTML и PHP:

`string fgetss (Int манипулятор, int длина [, string разрешенные_теги])`

Лекция 16 Авторизация доступа с помощью сессий

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.

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 16.1 Файл 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

Сессии - 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, SessionIdentifier). 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.

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()`;

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
?>

```

Листинг 16.2. 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_переменной';
```

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.)

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 16.3. 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 16.3b. 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.

Лекция 17 Регулярные выражения

Синтаксис патэрнов, используемый в этих функциях, очень напоминает Perl. Выражение должно быть заключено в ограничители, слэши (/), например. В качестве ограничителей могут использоваться любые символы, кроме не алфавитных символов, цифр и обратного слэша (\). Если символ ограничителя должен использоваться в самом выражении, он должен мнемонизироваться/escape обратным слэшем. Начиная с PHP 4.0.4, вы можете также использовать парные ограничители в стиле Perl: (), {}, [] и <>.

После конечного ограничителя могут идти различные модификаторы, влияющие на подстановку.

PHP также поддерживает регулярные выражения, использующие расширенный синтаксис POSIX.

Требования

Поддержка регулярных выражений предоставляется пакетом библиотеки PCRE, который является открытым ресурсом, автор Philip Hazel, и copyright University of Cambridge, England.

Установка

Начиная с PHP 4.2.0, эти функции по умолчанию включены. В более старых версиях вы должны конфигурировать и компилировать PHP с опцией

--with-pcre-regex[=DIR], чтобы использовать эти функции. Вы можете отменить функции pcre опцией --without-pcre-regex.

Конфигурация

Это расширение не определяет никаких директив конфигурации.

Типы ресурсов

Это расширение не определяет никакие типы ресурсов.

Пример 1. Примеры верных патэрнов

```
/<\/w+>/  
|(\d{3})-\d+|Sm  
/^(?i)php[34]/  
{^\s+(\s+)?$}
```

Пример 2. Примеры неправильных патэрнов

```
/href='(.*)' - отсутствует конечный ограничитель  
\/w+\s*\w+/J - неизвестный модификатор 'J'  
1-\d3-\d3-\d4| - пропущен начальный ограничитель
```

Отличия от Perl

Эти отличия даны относительно Perl 5.005.

По умолчанию пробельным символом считается любой таковой символ, распознаваемый функцией isspace() библиотеки C, хотя возможна компиляция PCRE с альтернативной таблицей типов символов. Нормально isspace() распознаёт space, formfeed, newline, carriage return, horizontal tab и vertical tab. Perl 5 больше не включает vertical tab в набор пробельных символов. Мнемоника \v, которая долгое время была в документации Perl, фактически никогда не распознавалась. Однако сам по себе этот символ рассматривался как пробельный как минимум до версии 5.002. В 5.004 и 5.005 он не соответствует \s.

PCRE не разрешает повторение квантификаторов или опережающих утверждений/lookahead assertions. Perl разрешает их, но они имеют другое значение. Например, (?!a){3} не утверждает, что три последующие символа не "a". Оно просто утверждает три раза, что следующий символ не "a".

Захваты субпатэрнов, которые возникают внутри отрицательного опережающего утверждения, вычисляются, но их вхождения в векторе смещения никогда не устанавливаются. Perl устанавливает свои числовые переменные из любого такого патэрна, который совпадает до неудачи утверждения при совпадении с чем-либо (продолжая таким образом), но только если негативное опережающее утверждение содержит одну ветвь.

Хотя двоичные нулевые символы поддерживаются в строке-субъекте, они не допускаются в строке патэрна, поскольку он передаётся как нормальная C-строка, оканчивающаяся нулём. Замена "\\x00" может использоваться в патэрне для представления бинарного нуля.

Следующие escape-последовательности (мнемоники) Perl не поддерживаются: \l, \u, \L, \U, \E, \Q.

Фактически они реализованы в общей обработке строк в Perl и не являются частью его машины патэрнов.

Perl-утверждение `\G` не поддерживается, так как оно не относится к одиночным совпадениям патэрна.

Вполне очевидно, что PCRE не поддерживает конструкцию `(?{code})`.

На момент написания в Perl 5.005_02 имелись некоторые странности, касающиеся установок захваченных строк, если часть патэрна повторяется. Например, сравнение "aba" с патэрном `/^(a(b)?)+$/` устанавливает в \$2 значение "b", но сравнение "aabbaa" с `/^(aa(bb)?)+$/` оставляет \$2 не установленной. Однако, если патэрн изменить на `/^(aa(b(bb))?)+$/`, то \$2 (и \$3) устанавливаются. В Perl 5.004 переменная \$2 устанавливается в обоих случаях и является TRUE в PCRE. Если в будущем Perl будет изменён для приведения этого в соответствие, PCRE может также измениться.

Другое пока не разрешённое несоответствие - то, что в Perl 5.005_02 патэрн `/^(a)?(?!(1)a|b)+$/` совпадает со строкой "a", а в PCRE - нет. Однако и в Perl, и в PCRE патэрн `/^(a)?a/` совпадает с "a" и оставляет \$1 не установленной.

PCRE предоставляет некоторые расширения к регулярным выражениям Perl:

Хотя смотрящие назад/lookbehind утверждения обязаны совпадать со строками фиксированной длины, каждая альтернативная ветвь lookbehind-утверждения может со строкой другой длины. Perl 5.005 требует, чтобы все они имели одну длину.

Если PCRE_DOLLAR_ENDONLY установлено, а PCRE_MULTILINE - нет, метасимвол \$ совпадает только с самым концом строки.

Если PCRE_EXTRA установлено, то обратный слэш с буквой, не имеющей специального значения, является ошибочным.

Если PCRE_UNGREEDY установлено, то жадность квантификаторов повторения инвертируется, то есть, по умолчанию они будут нежадными, но если за ними следует знак вопроса, будут жадными.

Субпатэрны

Субпатэрны ограничены скобками (круглыми), которые могут вкладываться. Маркировка части патэрна как субпатэрна выполняет два действия:

1. Локализует набор альтернатив. Например, патэрн `cat(aract|erpillar|)` совпадает с одним из слов: "cat", "cataract" или "caterpillar". Без скобок он совпадёт с "cataract", "erpillar" или с пустой строкой.

2. Устанавливает субпатэрн как захватывающий субпатэрн (как определено выше). Когда совпадает весь патэрн целиком, часть строки-субъекта, совпавшая с субпатэрном, передаётся обратно вызывающему посредством аргумента ovector функции `pcre_exec()`. Открывающие скобки вычисляются слева направо (начиная с 1) для получения количества захватывающих субпатэрнов.

Например, если строка "the red king" сопоставляется с патэрном ((red|white) (king|queen)) будут захвачены подстроки "red king", "red" и "king", и они будут пронумерованы 1, 2 и 3.

Фактически такое выполнение обычными скобками двух функций не всегда помогает. Бывают случаи, когда необходима группировка субпатэрнов без необходимости захвата. Если после открывающей скобки идёт "?:", субпатэрн не выполняет захвата и не учитывается при подсчёте количества захвативших субпатэрнов. Например, если строка "the white queen" сопоставляется с патэрном ((?:red|white)(king|queen)) то будут захвачены подстроки "white queen" и "queen", и они будут пронумерованы 1 и 2. Максимальное количество захватываемых подстрок - 99, а максимальное количество всех субпатэрнов, захватывающих и незахватывающих, равно 200.

В качестве удобной аббревиатуры, если любые установки опций нужны в начале незахватывающего субпатэрна, буквы опций могут появляться между "?" и ":". Таким образом, два субпатэрна (?i:saturday|sunday) (?:(?i)saturday|sunday) совпадают с одним и тем же набором строк. Поскольку альтернативные ветви пробуются слева направо, а опции не восстанавливают значения, пока не будет достигнут конец субпатэрна, установка опций в одной ветви не влияет на последующие ветви, и поэтому вышеприведённые патэрны совпадают с "SUNDAY", а также с "Saturday".

Повторение

Повторение специфицируется квантификаторами, которые могут идти после любого из следующих элементов:

одиноким символом, возможно, мнемонизированного метасимвола класса символов обратной ссылки/back reference (см. следующий раздел) субпатэрна в скобках (если это не утверждение/assertion, см. далее)

Квантификатор общего повторения специфицирует минимальное и максимальное количество допустимых совпадений, имея два числа в фигурных скобках, разделённые запятой. Число обязано быть менее 65536, а первое обязано быть меньше или равно второму. Закрывающая фигурная скобка сама по себе не является специальным символом. Если второе число отсутствует, но запятая есть, верхнего предела нет; если отсутствуют второе число и запятая, квантификатор специфицирует точное количество необходимых совпадений. Открывающая фигурная скобка, которая появляется в позиции, где квантификатор недопустим, или скобка, не соответствующая синтаксису квантификатора, считается литеральным символом. Например, {,6} это не квантификатор, а литеральная строка из 4 символов.

Квантификатор {0} допустим, заставляя выражение вести себя так, будто предыдущий элемент и квантификатор не существуют.

Для удобства (и обратной совместимости) три наиболее распространённых квантификатора имеют односимвольные сокращения:

* эквивалентен {0,}
+ эквивалентен {1,}
? эквивалентен {0,1}

Можно конструировать бесконечные циклы, введя после субпатэрна, который не совпадает ни с одним символом, квантификатор, не имеющий верхнего предела. Ранние версии Perl и PCRE являются источниками ошибок в процессе компиляции таких патэрнов. Однако, поскольку бывают случаи, когда это необходимо, такие патэрны принимаются, но если любое повторение такого субпатэрна фактически не совпадает ни с какими символами, цикл форсированно прерывается.

По умолчанию квантификаторы являются "жадными", то есть они совпадают максимально возможное количество раз (до максимально допустимого количества раз), не вызывая неудачи выполнения остальной части патэрна. Классический пример, когда это создаёт проблемы - попытка найти совпадения в комментарии C-программ. Комментарии появляются между символами /* и */, а внутри могут появляться отдельные символы *.

Если установлена опция PCRE_UNGREEDY (отсутствующая в Perl), то квантификаторы не жадничают по умолчанию, но отдельные могут быть жадными, если после них стоит знак вопроса. Другими словами, знак вопроса инвертирует поведение по умолчанию.

Когда субпатэрн в скобках квантифицирован минимальным количеством повторений, которое больше 1, или имеет ограничение максимума, для откомпилированного патэрна требуется больше места, пропорционально размеру минимума или максимума.

Если патэрн начинается с .* или с {0,} и установлена опция PCRE_DOTALL (эквивалентная Perl'овской /s), разрешая, таким образом совпадение . с символами новой строки, то патэрн неявно заякоривается, поскольку всё, что идёт следом, будет испытываться относительно каждой символьной позиции в строке-субъекте, поэтому после первой нет другой позиции для возобновления попыток найти полное совпадение. PCRE рассматривает такой патэрн так, как если бы ему предшествовало \A. Когда известно, что строка-субъект не содержит символов новой строки, предпочтительнее установить PCRE_DOTALL, если патэрн начинается с .*, чтобы получить эту оптимизацию, или, альтернативно, использовать ^ для явного обозначения заякоривания.

Когда захватывающий субпатэрн повторяется, захваченным значением является подстрока, которая совпадает с последней итерацией. Например, после того как (tweedle[dume]{3}\s*)+ совпадает с "tweedledum tweedledee", значением захваченной подстроки будет "tweedledee". Однако, если имеются вложенные захватывающие субпатэрны, соответствующие захваченные значения могут быть установлены в предыдущих итерациях.

Лекция 18 Использование шаблонов в PHP

Что такое шаблон в языке программирования? Можно сказать, что шаблон - это текст с переменными внутри него. При обработке шаблона происходит замена переменных на их значения.

В одной из лекций мы уже рассматривали пример шаблона. Это был шаблон для отображения документов. Пользователь создавал строку текста, размеченного с помощью html-тегов, и вставлял в нее специальные метасимволы (вида <!имя элемента>), которые наша программа впоследствии заменяла на значения соответствующих элементов. Для чего нам был нужен такой шаблон? Чтобы, например, можно было изменить стиль отображения документа, не меняя кода программы.

Наиболее распространенный ответ на вопрос, зачем нужны шаблоны, звучит примерно так: шаблоны нужны для того, чтобы отделить логику работы приложения от способа представления данных, т. е. от дизайна.

Приведенный пример шаблона - один из самых простых. Для его обработки используется только функция подстановки `str_replace()`. Чаще всего для того, чтобы работать с шаблонами, создают библиотеки классов. В принципе создавать свою библиотеку не обязательно, поскольку существует множество свободно распространяемых библиотек шаблонов, над функциональностью которых трудятся большие коллективы разработчиков, стараясь сделать их универсальными, мощными и быстрыми. Некоторые из таких библиотек мы и рассмотрим. Но для начала сформулируем задачу, на примере решения которой будем демонстрировать использование различных шаблонов.

Итак, задача:

Требуется сгенерировать web-страницу со списком статей, имеющих в базе данных. Для простоты считаем, что статья имеет название `title`, автора `author`, краткое содержание `abstract` и полное содержание `fulltext`, которое представлено либо в виде текста в базе данных, либо в виде ссылки на файл. Список должен быть организован так, чтобы при щелчке мышью на названии статьи ее полное содержание появлялось в новом окне.

Шаблоны подстановки

Как можно решить такую задачу способом простой подстановки, т.е. тем методом, которым мы решили задачу отображения документов?

Нужно придумать шаблон для этой страницы и где-то его хранить (в файле или в базе данных). Очевидно, что мы не можем придумать шаблон для всей страницы, потому что не знаем, сколько статей в базе данных. В шаблоне же мы договорились использовать только html и метасимволы <!имя элемента>. Поэтому мы можем написать только шаблон для одной строки списка, который уже программно надо преобразовать в нужное количество строк.

<a

```

href="/<!fulltext>"
target=new><!title></a>
(<!author>)<br><p>
<!abstract></p>

```

Кроме того, здесь есть еще одна загвоздка - с отображением ссылки на полный текст статьи. Если мы будем действовать по правилу подстановки (менять все метасимволы на их значения из базы данных), то может получиться, что вместо <!fulltext> вставим не ссылку на текст, а сам текст. То есть для этого элемента нужна дополнительная проверка перед заменой и какие-то дополнительные действия в случае, если в поле fulltext содержится текст статьи, а не ссылка на файл. Не будем усложнять себе жизнь и договоримся, что в поле fulltext всегда содержится только ссылка на файл. Тогда задачу можно решить следующим образом:

```

<?
$li_tmpl = file_get_contents("tmpl.html");
// считываем шаблон строки из файла
// устанавливаем соединение и выбираем
// базу данных
$conn = mysql_connect("localhost",
    "nina","123")
or die("Cant connect");
mysql_select_db("book");
$sql = "SELECT * FROM Articles";
$q = mysql_query($sql,$conn);
// отправляем запрос
$num = mysql_num_rows($q);
for($i=0; $i<$num; $i++){
    $tmpl .= $li_tmpl;
    $tmpl = str_replace("<!title>",
        mysql_result($q,$i,"title"),$tmpl);
    $tmpl = str_replace("<!author>",
        mysql_result($q,$i,"author"),$tmpl);
    $tmpl = str_replace("<!abstract>",
        mysql_result($q,$i,"abstract"),$tmpl);
    $tmpl = str_replace("<!fulltext>",
        mysql_result($q,$i,"fulltext"),$tmpl);
}
echo $tmpl;
?>

```

В принципе метод достаточно прост и удобен, но требует дополнительных усилий программиста при возникновении задач более сложных, чем простая подстановка значений. Для решения задач, где требуется делать подстановку целых блоков или даже проверять условия,

создают классы шаблонов, такие как FastTemplate и Smarty. Обсудим их подробнее.

Шаблоны FastTemplate

FastTemplate - это набор классов, позволяющих реализовать работу с шаблонами. Логика добавить в шаблон FastTemplate нельзя, вся она должна находиться в коде программы. Идея работы шаблонов FastTemplate заключается в том, что любая большая страница состоит из множества кусочков, самые маленькие из которых - обычные строки текста, и они получают имя и значение.

Что представляет собой файл шаблона FastTemplate? Это обычный html-файл, в котором могут встречаться переменные особого вида, впоследствии обрабатываемые методами класса FastTemplate.

Синтаксис переменных в шаблонах FastTemplate описывается следующим выражением: {[A-Z0-9_]+)}

Это значит, что переменная должна начинаться с фигурной скобки "{". Второй и последующие символы должны быть буквами верхнего регистра от A до Z, цифрами или символами подчеркивания. Переменная вычисляется с помощью закрывающей фигурной скобки "}":

```
{TITLE}
{AUTH20}
{TOP_OF_PAGE}.
```

Как уже было сказано, основная идея FastTemplate - создание страницы с помощью вложенных шаблонов. Например, для решения нашей задачи можно создать три файла шаблона:

main.tpl (Этот шаблон будет выводить страницу в целом):

```
<html>
<head><title>{TITLE_}</title>
</head>
<body>
{MAIN}
</body>
</html>
```

list.tpl (будет описывать, как выводить список в целом)

```
<ul>
{LIST_ELEMENT}
</ul>
```

list_element.tpl (описывает непосредственно элемент списка)

```
<li><a href="{FULLTEXT}">{TITLE}</a>
({AUTHOR})
<br> <p> {ABSTRACT}
```

Шаблоны мы создали - работу дизайнера выполнили. Теперь нужно научиться их обрабатывать, т.е. выполнить работу программиста. Сейчас создадим программу для обработки приведенных выше шаблонов.

Перед началом работы с шаблонами FastTemplate нужно подключить этот набор классов к нашей программе. В реальной жизни набор классов FastTemplate записан в один файл, как правило, с названием class.FastTemplate.php3, поэтому подключить его можно, например, с помощью команды:

```
include("class.FastTemplate.php3");
```

Следующий важный шаг - это создание объекта класса FastTemplate, с которым впоследствии мы будем работать:

```
$tpl = new FastTemplate(  
"/path/to/templates");
```

В качестве параметра передается путь к месту, где находятся наши шаблоны.

Методы FastTemplate

Далее необходимо изучить методы, которые можно применять к созданному объекту класса FastTemplate. Параллельно обратим внимание, как их можно использовать для решения нашей задачи.

Для работы с FastTemplate нужно знать четыре основных метода: define, assign, parse и FastPrint.

Метод define

Синтаксис:

```
define( array ( ключ => значение,  
ключ1 => значение1, ... ))
```

Метод define() связывает имя файла шаблона с более коротким именем, которое можно будет использовать в программе. То есть "ключ" - это имя, которое мы будем использовать в программе для ссылки на файл шаблона, имя которого записано в строке "значение". Реальные имена файлов шаблонов не рекомендуется использовать нигде, кроме метода define. При вызове метода define() происходит загрузка всех определенных в нем шаблонов:

```
$tpl->define( array (main => "main.tpl",  
list_f => "list.tpl",  
list_el=> "list_element.tpl" ));
```

Здесь мы задаем псевдонимы именам файлов шаблонов. Эти псевдонимы, т.е. переменные main, list_f и list_el, будут использоваться в программе вместо соответствующих имен файлов main.tpl, list.tpl и list_element.tpl.

Список использованных источников

1. Наварро Э. XHTML. Учебный курс. СПб.: Питер, 2001.
2. Тиге Дж.К. DHTML и CSS для Internet. М.: НТ Пресс, 2005.
3. Зельдман Д. Web-дизайн по стандартам. М.: НТ Пресс, 2005.
4. Дилип Найк Dynamic HTML: Стандарты и протоколы Интернета «Русская редакция» ТОО «Channel Trading Ltd.» 2005 г.
5. Джамса К. Эффективный самоучитель по креативному Web-дизайну. HTML, XHTML, CSS, JavaScript, PHP, ASP, ActiveX. Текст, графика, звук и анимация. М: "ООО ДиаСофтЮП", 2005.
6. В.Олифер, Н.Олифер. Компьютерные сети. Принципы, технологии, протоколы - СПб: "Питер", 2003.
7. Котеров Д. Самоучитель PHP 4. СПб.: БХВ, 2001.
8. Колесниченко Д.Н. Самоучитель PHP 5. СПб.: Наука и техника, 2004.
9. Мельников Д.А. Информационные процессы в компьютер.сетях. Протоколы, стандарты, интерфейсы, модели: М: КУДИЦ-ОБРАЗ, 2001, 256 с.
10. Романец Ю.В., Тимофеев П.А., Шаньгин В.Ф. Защита информации в компьютер.системах и сетях. Изд. 2-е, перераб., доп. М: Радио и связь, 2001, 376 с.