Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

Е. А. РУЖИЦКАЯ

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ПЛАТФОРМЕ NET: PHP.NET

Тексты лекций по спецкурсу для студентов специальности 1–40 01 01 «Программное обеспечение информационных технологий» специализации 1–40 01 01 01 «Компьютерные системы и Internet-технологии»

Гомель
УО «ГГУ им. Ф.Скорины»
2008

УДК 004.7+004.4(073)
ББК 32.973.232-018.2+32.973.26-018.2-2р30
Р 837

Рецензент:
кафедра вычислительной математики и программирования учреждения образования «Гомельский государственный университет имени Франциска Скорины»

Рекомендовано к изданию научно-методическим советом учреждения образования «Гомельский государственный университет имени Франциска Скорины».

Ружицкая, Е.А.
Р 837 Разработка программного обеспечения на платформе Net: PHP.Net : тексты лекций по спецкурсу «Разработка программного обеспечения на платформе Net: PHP.Net» для студентов специальности 1–40 01 01 «Программное обеспечение информационных технологий» специализации 1–40 01 01 01 «Компьютерные системы и Internet-технологии» / Е. А. Ружицкая; М-во обр. РБ, Гомельский государственный университет им. Ф. Скорины. – Гомель: ГГУ им. Ф.Скорины, 2008. – 135 с.

Тексты лекций содержат основы CGI-программирования, характеристику языка PHP, работу с данными формы, конструкции языка, стандартные функции работы с массивами и строками, файлами, рассмотрены механизмы работы сессий и работу с базой данных MySQL и адресованы студентам математических специальностей университета, а также могут быть использованы студентами экономических и физических специальностей, изучающих Internet-технологии.

УДК 004.7+004.4(073)
ББК 32.973.232-018.2+32.973.26-018.2-2р30

СОДЕРЖАНИЕ

Введение…… 4
Тема 1 Основы CGI-программирования…… 5
Тема 2 Характеристика языка PHP…….. 14
Тема 3 Работа с данными формы…… 23
Тема 4 Конструкции языка……29
Тема 5 Ассоциативные массивы…….. 34
Тема 6 Работа с массивами……42
Тема 7 Функции и области видимости…… 50
Тема 8 Строковые функции…….. 62
Тема 9 Математические функции…… 73
Тема 10 Работа с файлами и каталогами…… 76
Тема 11 Работа с датами и временем, посылка писем через PHP…… 94
Тема 12 Работа с www……98
Тема 13 Управление интерпретатором и сессиями…… 105
Тема 14 Работа с базой данных MYSQL…… 119
Тема 15 Загрузка файлов на сервер……. 130
Литература…… 134

ВВЕДЕНИЕ

PHP – мощный язык программирования, который позволяет создавать интерактивные web-сайты. Он хорошо работает на разнообразных платформах. My SQL – является одной из самых распространенных систем управления реляционными данными, используемой для создания высококачественных коммерческих баз данных.

Тексты лекций содержат основы CGI-программирования, характеристику языка PHP, работу с данными формы, конструкции языка, стандартные функции работы с массивами и строками, файлами, рассмотрены механизмы работы сессий и работу с базой данных MySQL и адресованы студентам математических специальностей университета, а также могут быть использованы студентами экономических и физических специальностей, изучающих Internet-технологии.

Тема 1 Основы CGI-программирования

1.1 Методы передачи данных
1.2 Передача документа пользователю
1.3 Передача информации CGI-сценарию

1.1 Методы передачи данных

Термин CGI (Common Gateway Interface – общий шлюзовой интерфейс) обозначает набор соглашений, которые должны соблюдаться Web-серверами при выполнении ими различных Web-приложений.

Программы, работающие в соответствии с соглашениями CGI, называют *сценариями.* Сценарии могут быть написаны на любом языке программирования. CGI – это механизм, который позволяет пользователю не только получать, но и передавать информацию серверу, а также формировать документы «на лету». Спецификация CGI описывает четыре набора механизмов обмена данными: через переменные окружения, командную строку, стандартный ввод, стандартный вывод.

Заголовки и метод GET. Способ посылки параметров сценарию, когда данные помещаются в командную строку URL, называется методом `GET`. Фактически, даже если не передается никаких параметров (например, при загрузке статической страницы), все равно применяется метод `GET`.

Рассмотрим механизм передачи данных в строкой броузера. Например, набираем в браузере строку `somestring`. Браузер анализирует строку, выделяет из нее имя сервера и порт (а также имя протокола), устанавливает соединение с Web-сервером по адресу `сервер:порт` и посылает ему что-то типа следующего:

```
GET somestring HTTP/1.0\n
...другая информация...
\n\n
```

Здесь `\n` означает символ перевода строки, а `\n\n` – два обязательных символа новой строки, которые являются маркером окончания заголовков запроса. Пока не пошлем этот маркер, сервер не будет обрабатывать запрос.

После GET-строки могут следовать и другие строки с информацией, разделенные символом перевода строки. Их обычно формирует браузер. Такие строки называются *заголовками* (headers), и их может быть сколько угодно. Протокол http задает правила формирования и интерпретации этих заголовков.

Итак, протокол http представляет собой набор заголовков, которыми обмениваются сервер и браузер. Не все заголовки обрабатываются сервером – некоторые просто пересылаются запускаемому сценарию

с помощью переменных окружения. *Переменные окружения* представляют собой именованные значения параметров, которые операционная система (точнее, процесс-родитель) передает запущенной программе. Программа может с помощью специальных функций получить значение любой установленной переменной окружения, указав ее имя.

Именно так и должен поступать CGI-сценарий, когда захочет узнать значение того или иного заголовка запроса. Однако набор передаваемых сценарию заголовков ограничен стандартами, и некоторые заголовки нельзя получить из сценария никаким способом (ему просто недоступна соответствующая переменная окружения).

Заголовки запросов и их описания.

- `GET`

  Формат: `GET сценарий?параметры HTTP/1.0`

  Переменные окружения: `REQUEST_URI`; в переменной `QUERY_STRING` сохраняется значение `параметры`, в переменной `REQUEST_METHOD` – ключевое слово `GET`.

  Этот заголовок является обязательным (если только не применяется метод `POST`) и определяет адрес запрашиваемого документа на сервере. Также задаются параметры, которые пересылаются сценарию (если сценарию ничего не передается, или же это обычная статическая страница, то все символы после знака вопроса и сам знак опускаются). Вместо строки `HTTP/1.0` может быть указан и другой протокол – например, `HTTP/1.1`. Именно его соглашения и будут учитываться сервером при обработке данных, поступивших от пользователя, и других заголовков. Строка `сценарий?параметры` задается в том же самом формате, в котором она входит в URL. Эта строка называется *URI* (Universal Resource Identifier – универсальный идентификатор ресурса).

  URL – это *полный* путь к некоторой Web-странице вместе с параметрами, а под URI понимается его *часть*, расположенная после имени (или IP-адреса) хоста и номера порта.

- `POST`

  Формат: `POST сценарий?параметры HTTP/1.0`

  Переменная окружения: `REQUEST_URI`; в переменной `QUERY_STRING` сохраняется значение `параметры`, в переменной `REQUEST_METHOD` – слово `POST`.

  Этот заголовок используется при передаче данных методом `POST`. Он отличается от метода `GET` тем, что данные можно передавать не только через командную строку, но и в конце всех заголовков.

  Сервер никак не интерпретирует POST-данные, а пересылает их непосредственно сценарию. Метод `POST` используется для передачи больших объемов данных, например, при загрузке файлов через Web

или при обработке больших форм. Кроме того, метод POST часто используют для эстетических целей: при применении GET URL сценария становится довольно длинным и неизящным, а POST-запрос оставляет URL без изменения, т.е. передаваемые данные не отображаются в командной строке.

- Content-type

  Формат: `Content-Type: application/x-www-form-urlencoded`

  Переменная: `CONTENT_TYPE`.

  Заголовок идентифицирует тип передаваемых данных. Обычно для этого указывается значение `application/x-www-form-urlencoded`, что означает формат, в котором все управляющие символы (отличные от алфавитно-цифровых и других отображаемых) специальным образом кодируются. Это тот самый формат передачи, который используется методами GET и POST. Довольно распространен формат `multipart/form-data`, который используется при загрузки файлов на сервер. Сервер не интерпретирует рассматриваемый заголовок, а просто передает его сценарию через переменную окружения.

- User-Agent

  Формат: `User-Agent: Mozilla/4.5 [en] (Win95; I)`

  Переменная окружения: `HTTP_USER_AGENT`.

  Уточняет версию браузера (в данном случае это Netscape Navigator).

- Referer

  Формат: `Referer: URL_адрес`

  Переменная окружения: `HTTP_REFERER`.

  Заголовок формируется браузером и содержит URL страницы, с которой осуществился переход на текущую страницу по гиперссылке.

- Content-length

  Формат: `Content-length: длина`

  Переменная окружения: `CONTENT_LENGTH`.

  Заголовок содержит строку, являющуюся десятичным представлением длины данных в байтах, передаваемых методом POST. Если задействуется метод GET, то этот заголовок отсутствует, и значит, переменная окружения не устанавливается.

- Cookie

  Формат: `Cookie: значения_Cookies`

  Переменная окружения: `HTTP_COOKIE`.

  Здесь хранятся все Cookies в URL-кодировке.

- Accept

Формат: `Accept: text/html, text/plain, image/gif, image/jpeg`

Переменная окружения: `HTTP_ACCEPT`.

В этом заголовке браузер перечисляет, какие типы документов он «понимает». Перечисление идет через запятую. Значение */* обозначает любой тип.

В методах GET и POST данные доставляются в URL-кодированном виде. Например, если нам нужно закодировать символ с шестнадцатеричным кодом 9E, это будет выглядеть так: %9E. Помимо этого, пробел представляется символом плюс (+).

1.2 Передача документа пользователю

Рассмотрим, как программа посылает свой ответ (то есть документ) пользователю. Сценарий просто помещает документ в *стандартный поток вывода* (на Си он называется `stdout`), который находится под контролем программного обеспечения сервера. Программа работает так, как будто нет никакого пользователя, а нужно вывести текст прямо на «экран». (Это она так думает, на самом деле выводимая информация будет перенаправлена сервером в браузер пользователя. Ясно, что у сценария никакого «экрана» нет и быть не может). Ответ программы, как и запрос пользователя, должен состоять из заголовков. Мы не можем просто направить документ в стандартный поток вывода: сначала нужно указать, в каком формате информация должна быть передана пользователю.

Заголовки ответа. Заголовки ответа должны следовать точно в таком же формате, как и заголовки запроса. А именно, это набор строк (завершающийся пустой строкой), каждая из которых представляет собой имя заголовка и его значение, разделенные двоеточием. Наличие пустого заголовка в конце также можно интерпретировать как два стоящих подряд обозначения `\n\n`. Затем следуют данные ответа, которые и являются документом, который будет отображен браузером.

Заголовок кода ответа. Здесь имеется одно отличие от формата, который используется в заголовках запроса. Первый заголовок ответа обязан иметь слегка специфичный вид – в нем не должно быть двоеточия. Он задает *код ответа сервера* и выглядит, например, так:

```
HTTP/1.1 OK
```

или так:

```
HTTP/1.1 404 File Not Found
```

В первом примере заголовок говорит браузеру, что все в порядке и дальше следует некоторый документ. Во втором примере сообщается, что затребованный файл не был найден на сервере.

Наиболее распространенные заголовки ответа:

- `Content-type`

Формат: `Content-type: mime_тип; charset=koi8-r`

Задает тип документа и его кодировку. Параметр `charset` задает кодировку документа. Поле `mime_тип` определяет тип информации, которую содержит документ: `text/html` – HTML-документ; `text/plain` – простой текстовый файл; `image/gif` – GIF-изображение; `image/jpeg` – JPG-изображение.

- `Pragma`

Формат: `Pragma: no-cache`

Запрещает кэширование документа браузером, так что при повторном визите на страницу браузер гарантированно загрузит ее снова, а не извлечет из своего кэша. Это может быть полезно, если страница содержит, например, динамический счетчик посещений.

- `Location`

Формат: `Location: http://www.host.com/page.html`

Заголовок определяет, что браузер пользователя должен перейти по указанному адресу, не дожидаясь тела документа ответа (как будто бы пользователь сам набрал в адресной строке нужный URL).

- `Set-cookie`

Формат: `Set-cookie: параметры_cookie`

Устанавливает Cookie в браузер пользователя.

- `Date`

Формат: `Date: Sat, 08 Jan 2000 11:56:26 GMT`

Указывает браузеру дату отправки документа.

- `Server`

Формат: `Server: Apache/1.3.9 (Unix) PHP/3.0.12`

Устанавливается сервером и указывает браузеру тип сервера и другую информацию о серверном программном обеспечении.

Пример CGI-сценария на Си:

```
#include <time.h> // Нужна для инициализации функции rand()
#include <stdio.h> // Поддержка функций ввода/вывода
#include <stdlib.h> // Поддержка функции rand()
// Главная функция. Именно она и запускается при старте сценария.
void main(void) {
// инициализируем генератор случайных чисел
int Num; time_t t; srand(time(&t));
// в Num записывается случайное число от 0 до 9
Num = rand()%10;
// далее выводим заголовки ответа. Тип — html-документ
printf("Content-type: text/html\n");
```



```
// запрет кэширования
printf("Pragma: no-cache\n");
// пустой заголовок
printf("\n");
// выводим текст документа — его мы увидим в браузере
printf("<html><body>");
printf("<h1>Здравствуйте!</h1>");
printf("Случайное число в диапазоне 0-9: %d",Num);
printf("</body></html>");
}
```

Исходный текст можно откомпилировать и поместить в каталог с CGI-сценариями на сервере. Обычно стараются все сценарии хранить в одном месте – в каталоге cgibin, у которого имеется разрешение на выполнение всех файлов внутри него.

Пусть нам нужен сценарий, который бы передавал пользователю какой-то GIF-рисунок. Делается это аналогично: выводим заголовок `Content-type: image/gif.` Затем копируем один-в-один нужный нам GIF-файл в стандартный поток вывода (лучше всего – функцией `fwrite`).

CGI-сценарии могут использоваться не только для вывода HTML-информации, но и для любого другого ее типа – начиная с графики и заканчивая звуковыми MIDI-файлами. Тип документа задается в единственном месте – заголовке `Content-type.`

1.3 Передача информации CGI-сценарию

Непосредственно перед запуском сценария сервер передает ему *переменные окружения,* в которых содержатся некоторые заголовки.

Переменные окружения:

- `HTTP_ACCEPT`

В этой переменной перечислены все MIME-типы данных, которые могут быть восприняты браузером. Строка `*/*` означает любой тип.

- `HTTP_REFERER`

Задает имя документа, в котором находится форма, запустившая CGI-сценарий.

- `HTTP_USER_AGENT`

Идентифицирует браузер пользователя. Если в данной переменной окружения присутствует подстрока `MSIE`, то это – Internet Explorer, в противном случае, если в наличии лишь слово `Mozilla`, – Netscape.

- `HTTP_HOST`

Доменное имя Web-сервера, на котором запустился сценарий.

- `SERVER_PORT`

Порт сервера (обычно 80), к которому обратился браузер пользователя.

- REMOTE_ADDR

Переменная окружения задает IP-адрес (или доменное имя) узла пользователя, на котором был запущен браузер.

- REMOTE_PORT

Порт, который закрепляется за браузером пользователя для получения ответа сервера.

- SCRIPT_NAME

Виртуальное имя выполняющегося сценария (то есть часть URL после имени сервера, но до символа ?).

- REQUEST_METHOD

Метод, который применяет пользователь при передаче данных.

- QUERY_STRING

Параметры, которые в URL указаны после вопросительного знака. Они доступны как при методе GET, так и при методе POST (если они были определены в атрибуте action тэга <form>).

- CONTENT_LENGTH

Количество байтов данных, присланных пользователем. Эту переменную необходимо анализировать, если нужно принять и обработать POST-формы.

Передача параметров методом GET. Все параметры передаются единой строкой (точно такой же, какая была задана в URL после ?) в переменной QUERY_STRING. Все данные поступят URL-кодированными. Для того чтобы узнать значения полученных переменных в Си, нужно воспользоваться функцией getenv().

Пример сценария на Си, работающего с переменными окружения:

```
#include <stdio.h> // Включаем функции ввода/вывода
#include <stdlib.h> // Включаем функцию getenv()
void main(void) {
// получаем значение переменной окружения REMOTE_ADDR
char *RemoteAddr = getenv("REMOTE_ADDR");
// ... и еще QUERY_STRING
char *QueryString = getenv("QUERY_STRING");
// печатаем заголовок
printf("Content-type: text/html\n\n");
// печатаем документ
printf("<html><body>");
printf("<h1>Здравствуйте. Я знаю все!</h1>");
printf("Ваш IP-адрес: %s<br>",RemoteAddr);
printf("Указанные параметры: %s",QueryString);
```



```
printf("</body></html>");
}
```

Откомпилируем сценарий и поместим его в «CGI-каталог». Теперь в адресной строке введем:

```
http://www.myhost.com/cgi-bin/script.cgi?a=1&b=2
```

Получим примерно такой документ:

```
Здравствуйте. Я знаю все!
Ваш IP-адрес: 192.232.01.23
Указанные параметры: a=1&b=2
```

Передача параметров методом POST. В отличие от метода GET, параметры передаются сценарию не через переменные окружения, а через *стандартный поток ввода* (в Си он называется stdin). То есть программа должна работать так, будто никакого сервера не существует, а она читает данные, которые вводит пользователь с клавиатуры.

Замечание: То, что был использован метод POST, вовсе не означает, что не был применен также и метод GET. Метод POST подразумевает также возможность передачи данных через URL-строку. Эти данные будут помещены в переменную окружения QUERY_STRING.

Для того, чтобы узнать, сколько именно данных переслал пользователь методом POST, служит переменная окружения CONTENT_LENGTH, в которой хранится строка с десятичным представлением числа переданных байтов данных (перед использованием ее надо перевести в обычное число).

Модифицируем предыдущий пример так, чтобы он принимал POST-данные, а также выводил и GET-информацию, если она задана:

Пример получения данных POST

```
#include <stdio.h>
#include <stdlib.h>
void main(void) {
// извлекаем значения переменных окружения
char *RemoteAddr = getenv("REMOTE_ADDR");
char *ContentLength = getenv("CONTENT_LENGTH");
char *QueryString = getenv("QUERY_STRING");
// вычисляем длину данных — переводим строку в число
int NumBytes = atoi(ContentLength);
// выделяем в свободной памяти буфер нужного размера
char *Data = (char *)malloc(NumBytes + 1);
// читаем данные из стандартного потока ввода
fread(Data, 1, NumBytes, stdin);
// добавляем нулевой код в конец строки
// (в Си нулевой код сигнализирует о конце строки)
Data[NumBytes] = 0;
```

```
// выводим заголовок
printf("Content-type: text/html\n\n");
// выводим документ
printf("<html><body>");
printf("<h1>Здравствуйте. Я знаю все!</h1>");
printf("Ваш IP-адрес: %s<br>",RemoteAddr);
printf("Количество          байтов          данных:
%d<br>",NumBytes);
printf("Указанные параметры: %s<br>",Data);
printf("А вот то, что мы получили через URL: %s",
QueryString);
printf("</body></html>");
}
```

Транслируем этот сценарий и запишем то, что получилось, под именем `script.cgi` в каталог, видимый извне как `/cgi-bin/`. Откроем в браузере следующий HTML-файл с формой:

Пример POST-формы:

```
<html><body>
<form action=/cgi-bin/script.cgi?param=value
method=post>
Name1: <input type=text name="name1"><br>
Name2: <input type=text name="name2"><br>
<input type=submit value="Запустить сценарий!">
</form>
</body></html>
```

Теперь, если набрать в полях ввода какой-нибудь текст и нажать кнопку, получим HTML-страницу, сгенерированную сценарием, например, следующего содержания:

```
Здравствуйте. Я знаю все!
Ваш IP-адрес: 136.234.54.2
Количество байтов данных: 23
Указанные перематры: name1=Vasya&name2=Petya
А вот то, что мы получили через URL: param=value
```

Обработка метода `POST` устроена сложнее, чем `GET`. Тем не менее, метод `POST` используется чаще, особенно если нужно передавать большие объемы данных или «закачивать» файл на сервер.

Если бы в предыдущем примере ввели параметры, содержащие, например, буквы кириллицы, то сценарию они бы поступили в URL-закодированном виде, поэтому дополнительно еще нужно было бы написать функцию расшифровки URL-кодированных данных. (Кодирование заключается в том, что некоторые неалфавитно-цифровые символы, в том числе и «русские» буквы, преобразуются в форму `%XX`, где `XX` – код символа в шестнадцатеричной системе счисления).

Тема 2 Характеристика языка PHP

2.1 Принцип работы PHP
2.2 Переменные, константы, выражения

2.1 Принцип работы PHP

PHP составлен из двух почти независимых блоков – транслятора и интерпретатора.

Транслятор – программа, которая переводит код с одного «языка» на другой. Например, утилита, преобразующая исходный Паскаль-код на Си – транслятор.

Компилятор – это транслятор, конвертирующий код программы на языке высокого уровня в машинный код.

Интерпретатор – это утилита, которая просматривает код некоторой программы и выполняет одну её инструкцию за другой, т.е. полностью контролирует процесс исполнения.

PHP, получая на свой вход исходный код программы, в первую очередь анализирует его (в частности, проверяет синтаксис) и *транслирует* во *внутреннее представление,* которое представляет собой специальный байт-код, с которым проще будет в дальнейшем оперировать PHP. Эту фазу чаще всего и называют ошибочно компиляцией. PHP исполняет (интерпретирует) полученный байт-код. В этот момент он представляет собой классический интерпретатор.

Таким образом, *PHP является интерпретатором с встроенным блоком трансляции, оптимизирующим ход интерпретации.*

Преимущества интерпретатора перед классическим компилятором, состоят в следующем:

1. Упрощается обнаружение ошибок во время выполнения программы. В случае сбоя интерпретатор сразу же выведет сообщение, если что-то не так.
2. Можно не заботиться об освобождении и выделении памяти. Интерпретатор сам определит, когда та или иная переменная в программе уже не используется, и освободит память, выделенную для нее.
3. Существует возможность написать программу, которая, грубо говоря, будет формировать и тут же исполнять другую программу. В частности, можно формировать идентификаторы во время исполнения программы, создавать массивы анонимных функций и т.д.
4. Не нужно думать о типах переменных.

Недостаток (единственный!): это медлительность интерпретаторов, даже с блоками трансляции. Проигрыш заметен в случае больших и сложных циклов, при обработке большого количества строк и т.д.

Примеры PHP-программ:

Первый вариант

```
<?
print ”Привет!”
?>
```

Второй вариант

```
<body>
Привет!
</body>
```

Третий вариант

```
<html>
<body>
   <h1> Привет!</h1>
<?
    //Вычисляем текущую дату в формате «день.месяц.год»
   $dat=date(”d.m.y”);
   $tm=date(”h:i:s”); //Вычисляем текущее время
   #Выводим их
   echo ”Текущая дата: $dat года <br>\n”;
   echo ”Текущее время: $tm <br>\n”;
   #Выводим цифры
   echo ”Квадраты и кубы первых 5 натуральных чи-
сел: <br>\n”;
   for ($i=1; $i<=5; $i++)
   {  echo ”<li>$i в квадрате = ”.($i*$i);
      echo ”, $i в кубе =”.($i*$i*$i).”\n”;
   }
?>
</body>
</html>
```

PHP-скрипт может не отличаться от обычного html-документа. Все, что расположено до начала PHP-кода, отображается непосредственно. Сам код сценария начинается после открывающегося тэг <? и заканчивается закрывающим ?>. Между этими двумя тэгами сам текст интерпретируется как программа, и в HTML-документ не попадает. Для вывода в программе используется оператор echo (это не функция, а конструкция языка) или `print`. PHP устроен так, что любой текст, который расположен вне программных блоков, ограниченных <? и ?>, выводится в браузер непосредственно, т.е. воспринимается, как вызов оператора echo.

2.2 Переменные, константы, выражения

Переменные. Имена переменных должны начинаться со знака $ и состоят из латинских букв и цифр. Имена переменных чувствительны к регистру ($my_variable, $My_variable). В PHP не нужно ни описывать переменные явно, ни указывать их тип. Интерпретатор делает это сам. Однако иногда он может ошибиться (например, если в текстовой строке на самом деле задано десятичное число), поэтому изредка возникает необходимость явно указывать тип переменной.

Переменные PHP это объекты, которые могут содержать в буквальном смысле все, что угодно (исключение – константа, которая может содержать только строку или число). Такого понятия как указатель в PHP не существует. При присваивании переменная копируется один-в-один, какую бы сложную структуру она не имела.

В программе не могут использоваться неинициализированные переменные!

Типы переменных. PHP непосредственно поддерживает 5 типов переменных:

- `integer` – целое число со знаком, обычно длиной 32 бита (от -2 147 483 648 до 2 147 483 647);
- `double` – вещественное число довольно большой точности;
- `string` – строка любой длины;
- `array` – ассоциативный массив (или его часто называют *хэш*). Это набор из нескольких элементов, каждый их которых представляет собой пару вида `ключ=>значение` (символом => обозначается соответствие определенному ключу какого-то значения). Доступ к отдельным элементам осуществляется указанием их ключа. В отличие от массивов Си ключами могут служить не только целые числа, начиная с нуля, но и любые строки. Оператор `array()` создает массив, элементы которого перечислены в скобках.

```
//  создаст массив с ключами ”0”, ”a”, ”b”, ”c”
$a=array(0=>”zzzz”, ”a”=>”aaa”,
         ”b”=>”bbb”, ”c”=>”ccc”);
echo $a[”b”]; //  выведет ”bbb”
$a[”1”]=”qq”; //создаст новый элемент в массиве
                //  и присвоит ему “qq”
$a[”a”]=”new_aaa”; //  присвоит существующему элементу
                    //  ”new_aaa”
```

- `object` – объект, реализующий несколько наиболее простых принципов объектно-ориентированного программирования. Внутренняя структура объекта похожа на хэш, за искючением того, что для доступа к отдельным элементам и функциям используется оператор `->`, а не квадратные скобки.

Логические переменные. Логическая переменная может содержать одно из двух значений: false или true. Кроме того, любое ненулевое число (и непустая строка) символизирует истину, тогда как 0, пустая строка – ложь.

```
echo false; //  выводит пустую строку
```

```
echo true; // выводит 1
```

Действия с переменными. Вне зависимости от типа переменной, с ней можно делать три основных действия.

Присвоение значения (оператор =).

Проверка существования. Можно проверить существует ли (то есть, инициализирована ли) указанная переменная. Осуществляется это с помощью оператора `IsSet()`. Если такая переменная существует, функция возвращает значение истина, в противном случае – ложно.

```
    if(IsSet($MyVar))
echo "Такая переменая есть. Её значение $MyVar";
```

Уничтожение. Уничтожение переменной реализуется оператором `Unset()`. После этого Действия переменная удаляется из внутренних таблиц интерпретатора, т.е. программы начинает выполняться так, как будто переменная еще не была инициализирована.

```
$a="Hello";
echo $a;
Unset($a); // теперь переменной а не существует
echo $a; // Ошибка: нет такой переменной
```

Применение `Unset()` для работы с обычными переменными нецелесообразно. Она используется для удаления элементов в ассоциативном массиве. Например, если в массиве `$A` нужно удалить элемент с ключом `for_del`:

```
    Unset($A["for_del"]);
```

Стандартные функции определения типа переменной:

```
is_integer($a)  //возвращает true, если $a – целое число
is_double($a)   //возвращает true,
                //если $a – действительное число
is_string($a)   //возвращает true, если $a – строка
is_array($a)    //возвращает true, если $a – массив
is_object($a)   //возвращает true, если $a – объект
is_boolean($a)  //возвращает true,
                //  если $a – логическая переменная
is_gettype($a)  //возвращает строки со значениями array, object,
                //integer, double, string, boolean или unknown type
                //в зависимости от типа переменной. Последнее
                //значение возвращается для тех переменных,
                //типы которых не являются встроенными в PHP.
```

Установка типа переменной

Функция `settype(переменная,тип)` пытается привести тип указанной переменной к одному из стандартных типов и возвращает значение false, если это не удалось.


```
    settype($a,"integer");
```

Оператор присваивания

```
$имя_переменной=значение;
```

В конце каждого оператора должна стоять ;

Ссылочные переменные. Хотя в PHP нет такого понятия как указатель, все же можно создавать ссылки на другие переменные. Существует две разновидности ссылок: жесткие и символические.

Жесткая ссылка представляет собой просто переменную, которая является синонимом другой переменной. Многоуровневые ссылки (то есть ссылка на ссылку) не поддерживаются. Жесткие ссылки создаются с помощью оператора &.

```
$a=10;
$b=&$a; // теперь $b то же самое, что и $a
```

Жесткая ссылка и переменная (объект), на которую она ссылается, совершенно равноправны, изменение одной влечет изменение другой. Оператор `Unset()` выполненный для жесткой ссылки, не удаляет объект, на которую она ссылается, а всего лишь разрывает связь между ссылкой и объектом. Объект удаляется только тогда, когда на него никто уже не ссылается. Жесткие ссылки применяются при передаче параметров функции и возврате значения из нее.

Символическая ссылка – строковая переменная, хранящая имя другой переменной. Чтобы добраться до значения переменной, на которую ссылается символическая ссылка, необходимо применить оператор разыменования – дополнительный знак $ перед именем ссылки.

```
$a=10;
$p="a";   //  присваиваем $p имя другой перемнной
echo $$p;//выводим переменную, на которую ссылается $p, т.е. $a
$$p=100; //  присваивает $a значение 100
```

Итак, для того, чтобы использовать обычную строковую переменную как ссылку, нужно перед ней поставить еще один знак $. Это говорит интерпретатору, что надо взять не значение самой $p, а значение переменной, имя которой хранится в переменной $p.

Использование символических ссылок лучший способ запутать программу, поэтому старайтесь их избегать!

Константы. Константа отличается от переменной тем, что ей нигде в программе нельзя присвоить значение более одного раза, и её имя не предваряется знаком $, как это делается для переменных. Константы бывают 2 типов: предопределенные (установленные самим интерпретатором) и определенные программистом.

Предопределенные константы:

`_FILE_` – хранит имя файла программы, которая выполняется в данный момент

LINE – содержит текущий номер строки, которую обрабатывает в текущий момент интерпретатор

PHP_VERSION – версия интерпретатора PHP

PHP_OS – имя операционной системы, под которой работает PHP

TRUE или true – истина

FALSE или false – ложно

Определенные пользователем константы задаются при помощи оператора define()

```
define(”pi”,3.14);
define (”str”,”Hello”);
echo sin(pi/4);
echo str;
```

Функция defined(имя константы) проверяет, существует ли константа с указанным именем и возвращает значение false, если такой константы нет.

Выражения

Числовые выражения

```
$a=5;                    // a=5
$a=($b=10);              // $a=$b=10
$a=4*sin($b=$c+10)+$d;   // $b=$c+10
                         // $a=4*sin($c+10)+$d
```

Следующие операторы преобразования типов используются как в функциональной, так и в префиксной операторной форме, т.е. следующие конструкции эквивалентны:

```
$a=intval($b);
$a=(int)$b;
```

Операторы преобразования типов:

$b=intval(выражение) или $b=(int) (выражение)

// переводит выражение в целое число

$b=doubleval(выражение) или $b=(double) (выражение)

// переводит выражение в действительное число

$b=strval(выражение) или $b=(string) (выражение)

// переводит выражение в строку

$b=(bool) (выражение)

// переводит выражение в логический тип.

Логические выражения. Логические операторы: <, > , == (равно), || (или), ! (не), && (и). Переменным можно присваивать значения логических выражений:

```
$a=10<5;              //  a=false
$a=$b==1;             //  $a=true, если $b=1
$a=$b>1&&$b<=10       //  $a=true, если $b в пределах от1 до 10
$a=!($b||$c)&&$d;     //$a=true, если $b и $c ложны, а $d – истинно
```



```
$b=$a>=1&&a<=10;   //присваиваем $b значение логического
                   // выражения
if ($b) echo ”a в нужно диапазоне значений”;
```

Строковые выражения. Строки могут содержать текст вместе с символами форматирования. Определение строки в кавычках или апострофах может начинаться на одной строке, а завершаться на другой.

Строка, заключенная в апострофы трактуется так же, как записана, за исключением двух последовательностей символов:

\' трактуется как апостроф, предназначенный для вставки апострофа в строку, заключенную в апострофы;

\\ трактуется как обратный слэш и позволяет вставить в строку этот символ.

Строки в кавычках могут содержать набор специальных метасимволов:

\n – символ новой строки;

\r – символ возврата каретки;

\t – символ табуляции;

\$ – обозначает символ $, чтобы следующий за ним текст не был интерпретирован как переменная;

\” – кавычка;

\\ – обратный слэш;

\xNN символ с шестнадцатеричным кодом NN.

Для слияния строк используется операция конкатенации “.”. Операторы, выводящие текстовые строки.

```
echo ”Текущая дата: $dat года <br>\n”;
echo ”Текущая дата:”,$dat,” года <br>\n”;
echo ”Текущая дата:”.$dat.” года <br>\n”;
```

Плюс используется как числовой оператор, а точка – как строковый:

```
$a=”100”;
$b=”200”;
echo $a+$b; //  выведет “300”
echo $a.$b; //  выведет “100200”
```

Here–документ. Это способ записи строковых констант, который представляет собой альтернативу для записи многострочных констант.

```
$a=<<<MARKER
Далее идет какой-то текст, возможно,
с переменными $name значения которых будут
вставлены в документ
MARKER;
```

Строка MARKER может быть любым алфавитно-цифровым идентификатором. Синтаксис накладывает 2 ограничения на here-документы:

1) после <<<MARKER и до конца строки не должны идти никакие непробельные символы;

2) завершающая строка MARKER; должна оканчиваться точкой с запятой, после которой до конца строки не должно быть никаких инструкций. MARKER – это любой идентификатор.

Вызов внешней программы. Строка в *обратных апострофах* заставляет PHP выполнить команду операционной системы и то, что она вывела, подставить на место строки в обратных апострофах. Например, содержимое текущего каталога можно узнать так:

```
$st='command.com/c dir';
echo "<pre>$st</pre>";
```

Операции:

Арифметические: +, -, *, /, a%b (остаток от делении a на b, работает только с целыми числами), $a+=4; $b-=3;

Строковые: a.b – слияние строк; a[n] – символ строки в позиции n.

Присваивания (=).

Инкремента и декремента ($a++, $a--, --$a, ++$a).

Битовые операции. Предназначены для работы (установки/снятия/проверки) групп битов целой переменной. Биты целого числа – это отдельные разряды того же самого числа, записанного в двоичной системе счисления: a&b, a|b, `a, a<<b (сдвиг влево), a>>b (сдвиг вправо).

Операции *сравнения*: a==b, a!=b, a<b, a>b, a<=b, a>=b.

Операции *эквивалентности*: это тройной знак равенства = = =, или оператор проверки на эквивалентность. Это оператор не только сравнивает выражения, но и их типы. Есть антипод этого оператора (!= =) – не эквивалентны.

Логические операции. !a, a&&b, a||b

Оператор отключения предупреждений. В PHP-программах все ошибки выводятся в окно браузера вместе с указанием строки и файла, в которых они находятся. PHP ранжирует ошибки и предупреждения по четырем основным «уровням серьезности». Можно настроить PHP так, чтобы он выдавал только ошибки тех уровней, которые необходимы. Для отключения выдачи ошибок в окно браузера, существует оператор @(отключение ошибок). Если разместить этот оператор перед любым выражением, то сообщения об ошибках в этом выражении будут подавлены и в окне броузера не отображены.

```
<? if (@$doGo) echo "Вы нажали кнопку!"; ?>
```


Виды комментариев:

операторы // комментарий

однострочный комментарий

/*

многострочные комментарии (ими *лучше не пользоваться!* так как они вступают в конфликт с русскими буквами)

*/

Замечания

1 Для вывода значений переменных, переданных из форм, нужно использовать оператор `print` и указывать метод передачи данных:

```
print "Ваше имя:".$_GET['name'];
print "Ваше имя:".$_POST['name'];
```

2 *Условные обозначения при использовании стандартных функций PHP:*

- `string` – обычная строка, или тип, который можно перевести в строку;
- ***`int, long`*** – целое число, либо вещественное число (в последнем случае дробная часть отсекается), либо строка, содержащая число в одном из перечисленных форматов. Если строку не удается перевести в int, то вместо нее подставляется 0, и никаких предупреждений не генерируется!
- ***`double, float`*** – вещественное число, или целое число, или строка, содержащая одно из таких чисел.;
- `bool` – логический тип, который будет восприниматься либо как ложь (нулевое число, пустая строка или константа false), либо как истина (все остальное);
- `array` – ассоциативный массив;
- `list` – массив с целыми ключами, пронумерованными от 0 и следующими подряд. Так как список является разновидностью ассоциативного массива, то обычно вместо параметров функций типа `list` можно подставлять и параметры типа `array`;
- `object` – объект какой-то структуры;
- `void` – применяется только для определения возвращаемого функцией значения (не возвращает ничего);
- `mixed` – все, что угодно. Это может быть целое или дробное число, строка, массив или объект.

Тема 3 Работа с данными формы

3.1 Передача данных
3.2 Трансляция полей формы в переменные

3.1 Передача данных

Web-программирование в большей части (или хотя бы наполовину) представляет собой обработку различных данных, введенных пользователем – т. е., обработку форм.

Передача данных командной строкой. Напишем сценарий, который принимает в параметрах Имя и возраст пользователя и выводит: "Привет, <имя>! Я знаю, вам <возраст> лет!".

Рассмотрим способ передачи имени и возраста сценарию – непосредственный набор их в URL после знака ? – например, в формате

http://www.script.php?name=имя&age=возраст

Пусть на сервере в корневом каталоге есть сценарий на PHP под названием `hello.php`. Наш сценарий распознает 2 параметра: `name` и `age`. Он должен отработать и вывести следующую HTML-страницу:

```
<html>
<body>
Привет, name! Я знаю, Вам age лет!
</body>
</html>
```

Нужно `name` и `age` заменить на соответствующие значения. Таким образом, если задать в адресной строке браузера

http://www.somehost.com/script.php?name=Vasya&age=20

мы должны получить страницу с требуемым результатом.

Чтобы в сценарии получить строку параметров, переданную после знака вопроса в URL при обращении к сценарию, можно проанализировать переменную окружения QUERY_STRING, которая в PHP доступна под именем $QUERY_STRING.

```
<html>
<body>
<?
print "Данные из командной строки: $QUERY_STRING";
?>
</body>
</html>
```

Если теперь мы запустим этот сценарий из браузера (перед этим сохранив его в файле `test.php` в корневом каталоге сервера) примерно следующим образом:

http://www.myhost.com/test.php?aaa+bbb+ccc+ddd

то получим документ следующего содержания:

Данные из командной строки: aaa+bbb+ccc+ddd

Обратите внимание на то, что URL-декодирование символов не произошло: строка $QUERY_STRING, как и одноименная переменная окружения, всегда приходит в той же самой форме, в какой она была послана браузером.

Так как PHP изначально создавался именно как язык для Web-программирования, то он дополнительно проводит некоторую работу с переменной $QUERY_STRING перед тем, как управление будет передано сценарию, а именно, он разбивает ее по пробельным символам (в нашем примере пробелов нет, их заменяют символы +, но эти символы PHP также понимает правильно) и помещает полученные кусочки в массив-список `$argv`, который впоследствии может быть проанализирован в программе.

Однако массив `$argv` используется при программировании на PHP крайне редко, что связано с гораздо большими возможностями интерпретатора по разбору данных, поступивших от пользователя. Однако в некоторых ситуациях его применение оправдано.

Формы. Для того, чтобы пользователь мог в удобной форме ввести свое имя и возраст, нужно создать обычный HTML-документ (например, `form.html`) с элементами этого диалога – текстовыми полями – и кнопкой, т.е. форму:

```
<html>
<form action=hello.php>
Введите имя:
<input type=text name="name" value="Елена"><br>
Ваш возраст:
<input type=text name="age" value="5"><br>
<input type=submit value="Отправить">
</form>
</body>
</html>
```

Загрузим наш документ в браузер. Теперь, если ввести в поле с именем свое имя, а в поле для возраста – свой возраст и нажать кнопку, браузер автоматически обратится к сценарию `hello.php` и передаст через ? все атрибуты, расположенные внутри тэгов `<INPUT>` в форме и разделенные символом & в строке параметров. Заметьте, что в атрибуте `action` тэга `<form>` мы задали относительный путь, т.е. сценарий `hello.php` будет искаться браузером в том же самом каталоге, что и файл `form.html`.

Все перекодирования и преобразования, которые нужны для URL-кодирования данных, осуществляются браузером автоматически. В ча-

стности, буквы кириллицы превратятся в %xx, где xx – шестнадцатеричное число, обозначающее код символа.

Листинг hello.php – модель простого PHP-сценария

```
<html>
<body>
<?
print "Привет, ".$_GET['name'].
      ". Вам ".$_GET['age']." лет.";
?>
</body>
</html>
```

3.2 Трансляция полей формы в переменные

Интерпретатор все данные из полей формы преобразует в глобальные *одноименные* переменные. Значение поля `name` после начала работы программы будет храниться в переменной `$name`, а значение поля `age` – в переменной `$age`.

Теперь сделаем так, чтобы при запуске без параметров сценарий выдавал документ с формой, а при нажатии кнопки – выводил нужный текст. Самый простой способ определить, был ли сценарий запущен без параметров – проверить, существует ли переменная с именем, совпадающим с именем кнопки отправки. Если такая переменная существует, то, очевидно, что пользователь запустил программу, нажав на кнопку.

```
<html>
<body>
<?if(!@$doGo)
{
?>
<form action="<?=$SCRIPT_NAME?>">
//<form action=hello.php>
Введите имя:
<input type=text name="name" value="Елена"><br>
Ваш возраст:
<input type=text name="age" value="5"><br>
<input type=submit name="doGo" value="Отправить">
</form>
<?
} else {
?>

Привет!<br>
<?
```



```
print "Привет, ".$_GET['name'].
      ". Вам ".$_GET['age']." лет.";
?>
<?
}
?>
</body>
</html>
```

Конструкция `<?=выражение?>` является более коротким обозначением для `<?echo(выражение) ?>`, и предназначена для того, чтобы вставлять величины прямо в HTML-страницу.

Обратите внимание на полезный прием: в параметре `action` тэга `<form>` мы не задали явно имя файла сценария, а извлекли его из переменной `SCRIPT_NAME` (которая устанавливается автоматически перед запуском сценария). Это позволило нам не "привязываться" к имени файла, т.е. теперь мы можем его в любой момент переименовать без потери функциональности.

Если PHP установлен не как модуль Apache, а как отдельный обработчик, то переменная `$SCRIPT_NAME` будет содержать не то значение, на которое мы рассчитываем.

К тому же, теперь исчезла необходимость и в промежуточном файле form.html, его код встроен в сам сценарий.

Трансляция переменных окружения и Cookies. В переменные преобразуются не только все данные формы, но и переменные окружения (включая `QUERY_STRING`, `CONTENT_LENGTH` и многие другие), а также все `Cookies`.

Пример сценария, который печатает IP-адрес пользователя, который его запустил, а также тип его браузера (эти данные хранятся в переменных окружения `REMOTE_USER` и `HTTP_USER_AGENT`):

```
<html><body>
Ваш IP-адрес: <?=$REMOTE_USER?><br>
Ваш браузер: <?=$HTTP_USER_AGENT?>
</body></html>
```

По умолчанию, трансляция выполняется в порядке ENVIRONMENT-GET-POST-COOKIE, причем каждая следующая переменная как бы перекрывает предыдущее свое значение. Например, пусть у нас есть переменная окружения `a=10`, параметр, поступивший из `GET`-формы `a=20` и `Cookie a=30`. В этом случае в переменную `$a` сценария будет записано 30, поскольку `Cookie` перекрывает `GET`, а `GET` перекрывает переменные окружения.

Трансляция списков. Механизм трансляции полей формы в PHP работает приемлемо, когда среди них нет полей с одинаковыми имена-

ми. Если же таковые встречаются, то в переменную записываются только данные последнего встретившегося поля. Это довольно-таки неудобно при работе, например, со списком множественного выбора <select multiple>:

```
<select name=Sel multiple>
<option>First
<option>Second
<option>Third
</select>
```

В таком списке вы можете выбрать (подсветить) не одну, а сразу несколько строчек, используя клавишу <Ctrl> и щелкая по ним кнопкой мыши. Пусть мы выбрали First и Third. Тогда после отправки формы сценарию придет строка параметров `Sel=First&Sel=Third`, и в переменной `$Sel` окажется только `Third`. Для решения подобных проблем в PHP предусмотрена возможность давать имена полям формы в виде имени массива с индексами:

```
<select name="Sel[]" multiple>
<option>First
<option>Second
<option>Third
</select>
```

Теперь сценарию придет строка `Sel[]=First&Sel[]=Third`, интерпретатор обнаружит, что мы хотим создать "автомассив" (то есть массив, который не содержит пропусков, и у которого индексация начинается с нуля), и, действительно, создаст переменную `$Sel` типа массив, содержимое которого следующее:

```
array (0=>"First", 1=>"Third").
```

Пример работы с автомассивами:

```
$A[]=10;
$A[]=20;
$A[]=30;
```

После отработки этих строк будет создан массив `$A`, заполненный последовательно числами 10, 20 и 30, с индексами, отсчитываемыми с нуля. То есть, если внутри квадратных скобок при присваивании элементу массива не указано ничего, то подразумевается элемент массива, следующий за последним.

Прием с автомассивом применим не только к этому элементу формы: автомассивы мы можем применять и в любых других полях.

Пример, создающий 2 переключателя (кнопки со значениями вкл/выкл), один редактор строки и одно текстовое (многострочное) поле, причем все данные после запуска сценария, обрабатывающего эту форму, будут представлены в виде одного-единственного автомассива:


```
<input type=checkbox name=Arr[] value=ch1>
<input type=checkbox name=Arr[] value=ch2>
<input type=text name=Arr[] value="Some string">
<textarea name=Arr[] >Some text</textarea>
```

Автомассивы можно использовать для любых элементов формы.

Трансляция массивов. Пусть имеется форма, содержащая следующие элементы:

```
Имя: <input type=text name=Data[name]><br>
Адрес: <input type=text name=Data[address]><br>
Город:<br>
<input type=radio name=Data[city]
       value=Moscow>Москва<br>
<input type=radio name=Data[city]
       value=Peter>Санкт-Петербург<br>
<input type=radio name=Data[city]
       value=Kiev>Киев<br>
```

После передачи подобных данных сценарию на PHP в нем будет инициализирован ассоциативный массив `$Data` с ключами `name`, `address` и `city`. То есть, имена полям формы можно давать не только простые, но и представленные в виде одномерных ассоциативных массивов.

В сценарии к отдельным элементам формы можно будет обратиться при помощи указания ключа массива: например, `$Data['city']` обозначает значение той радиокнопки, которая была выбрана пользователем, а `$Data["name"]` – введенное имя. В сценарии мы обязательно должны заключать ключи в кавычки или апострофы – в противном случае интерпретатором будет выведено предупреждение. В то же время, в параметрах name полей формы мы, наоборот, должны их избегать. Многомерные массивы (то есть, массивы массивов) указывать нельзя.

Тема 4 Конструкции языка

4.1 Условные инструкции
4.2 Инструкции циклов
4.3 Инструкции включения

4.1 Условные инструкции

О терминологии. Термины «конструкция» и «инструкция» совершенно эквивалентны. Термины «оператор» и «операция» несут разную смысловую нагрузку: любая операция есть оператор, но не наоборот. Например, echo – оператор, но не операция, а ++ – операция.

Инструкция if–else.

Формат

```
if (условие)
    инструкция_1;
else
    инструкция_2;
```

Альтернативный синтаксис

```
if (условие1):
    команды;
elseif (условие2):
    другие__команды;
else:
    иначе_команды;
endif
```

Конструкция `else` может опускаться.

```
if ($a>=1&&$b<=10) echo "Все ОК";
else echo "Неверное значение в переменной!";
```

Если инструкция_1 или инструкция_2 должны состоять из нескольких команд, то они, заключаются в фигурные скобки:

```
if ($a>$b) { print "a больше b"; c=$b; }
else if {$a==$b) { print "a равно b"; $c=$a; }
     else { print "a меньше c"; $c=$a; }
```

Замечание: `elseif` пишется слитно! Обратите внимание на расположение двоеточия (:)! Если его пропустить, будет сгенерировано сообщение об ошибке. Блоки `elseif` и `else` можно опускать.

Использование альтернативного синтаксиса. Для того чтобы вставить HTML-код в тело сценария, достаточно просто закрыть скобку ?>, написать этот код, а затем снова открыть ее при помощи <?, и продолжать программу.

Чаще всего, однако, нужно бывает делать не вставки HTML внутрь программы, а вставки кода внутрь HTML. Поэтому целесообразно бывает отделять HTML-код от программы, например, поместить его в отдельный файл, который затем подключается к программе при помощи инструкции `include`.

Пример сценария с использованием альтернативного синтаксиса `if-else`:


```
<?if(@$go):?>
Привет, <?=$name?>!
<?else:?>
<form action=<?=$REQUEST_URL?> method=post>
Ваше имя: <input type=text name=name><br>
<input type=submit name=go value="Отослать!">
<?endif?>
```

Конструкция switch-case.

Формат

```
switch (выражение)
{
case значение1: команды1;
                [break;]
case значение2: команды2;
                [break;]
…
case значениеN: командыN;
                [break;]
[default:
команды_по_умолчанию;
[break]]
}
```

Альтернативный синтаксис

```
switch (выражение):
case значение1: команды1;
                [break;]
case значение2: команды2;
                [break;]
…
case значениеN: командыN;
                [break;]
[default:
 команды_по_умолчанию;
[break]]
endswitch;
```

4.2 Инструкции циклов

Цикл с предусловием while.

Формат

```
while (условие)
    инструкция;
```

Альтернативный синтаксис

```
while (условие):
    команды;
endwhile;
```

Цикл с постусловием do-while.

```
do
{   команды; }
while (логическое_выражение);
```

Альтернативного синтаксиса для `do-while` нет.

Универсальный цикл for.

Формат

```
for
(инициализирующие_команды;
условие_цикла;
команды_после_прохода)
  тело_цикла;
```

Альтернативный синтаксис

```
for
(инициализируюшие__команды;
условие_цикла;
команды_после_прохода):
операторы;
endfor;
```

Инструкции break и continue. Инструкция `break` осуществляет немедленный выход из цикла. Она может задаваться с одним необязательным параметром – числом, которое указывает, из какого вложенного цикла должен быть произведен выход. По умолчанию используется 1, т. е. выход из текущего цикла, но иногда применяются и другие значения:

```
for ($i=0; $i<10; $i++)
{
    for($j=0; $j<10; $j++)
      If($A[$i]==$A[$j]) break(2);
}
If ($i<10)
 echo "Есть совпадающие элементы в матрице A!";
```

В этом примере инструкция `break` осуществляет выход не только из второго, но и из первого цикла, поскольку указана с параметром 2.

Инструкция `continue` так же, как и `break`, работает только «в паре» с циклическими конструкциями. Она немедленно завершает текущую итерацию цикла и переходит к новой. Так же, как и для `break`, для `continue` можно указать уровень вложенности цикла, который будет продолжен по возврату управления.

Цикл foreach. Данный тип цикла предназначен специально для перебора всех элементов ассоциативного массива. Формат:

`foreach (`массив `as $key=>$value)`
команды;

Здесь команды циклически выполняются для каждого элемента массива, при этом очередная пара `ключ=>значение` оказывается в переменных `$key` и `$value`.

У цикла `foreach` имеется и другая форма записи, которую следует применять, когда нас не интересует значение ключа очередного элемента. Выглядит она так:

`foreach (`массив `as $value)`
команды;

В этом случае доступно лишь *значение* очередного элемента массива, но не его ключ. Такой цикл применяется для работы с массивами-списками.

Цикл `foreach` оперирует не исходным массивом, а его *копией.* Это означает, что любые изменения, которые вносятся в массив, не могут быть «видны» из тела цикла. Что позволяет, например, в качестве массива использовать не только переменную, но и результат работы какой-нибудь функции, возвращающей массив (в этом случае функция будет вызвана всего один раз – до начала цикла, а затем работа будет производиться с копией возвращенного значения).

4.3 Инструкции включения

Инструкция require. Эта инструкция позволяет разбить текст программы на несколько файлов:

`require` имя_файла;

При запуске (именно при запуске, а не при исполнении!) программы интерпретатор заменит инструкцию на содержимое файла имя_файла (этот файл может также содержать сценарий на PHP, обрамленный, как обычно, тэгами <? и ?>). Причем сделает он это *только один раз* (в отличие от `include`): а именно, непосредственно перед запуском программы. Эта инструкция используется например, для включения в вывод сценария «шапок» с HTML-кодом.

```
<?
require "header.htm";
```
. . . работает сценарий и выводит само тело документа
```
require "footer.htm";
?>
```

Инструкция include. Эта инструкция практически идентична `require`, за исключением того, что включаемый файл вставляется тело сценария не перед его выполнением, а прямо во время.

Пусть у нас есть 10 текстовых файлов с именами file0.php, file1.php и так далее до file9.php, содержимое которых просто десятичные цифры 0, 1 ... 9 (по одной цифре в каждом файле). Запустим такую программу:

```
for($i=0; $i<10; $i++)
{
    include "fiie$i.php";
}
```

В результате получим вывод, состоящий из 10 цифр: «0123456789». Каждый из файлов был включен по одному разу прямо во время выполнения цикла!

Фигурные скобки вокруг `include` *не являются необязательными*. Если их убрать то будет выдано сообщение об ошибке (или программа начнет неправильно работать). Это происходит потому, что `include` не является на самом деле оператором в привычном нам смысле этого слова. Каждый раз, когда интерпретатор встречает инструкцию `include`, он просто заменяет ее на содержимое файла, указанного в параметре. Если в этом файле несколько команд, то в цикле выполнится только первая из них, а остальные будут запущены уже *после* окончания цикла. Поэтому необходимо всегда обрамлять инструкцию `include` фигурными скобками, если размещаете ее внутри какой-либо конструкции.

Трансляция и проблемы с include. Перед исполнением PHP транслирует программу во внутреннее представление. Это означает, что в памяти создается как бы «полуфабрикат», из которого исключены все комментарии, лишние пробелы, некоторые имена переменных и т.д. Впоследствии это внутреннее представление интерпретируется (выполняется). Однако в программе могут встретиться такие места для интерпретатора, которые PHP не сможет оттранслировать заранее. В этом случае он их пропускает, чтобы в момент, когда управление дойдет до определенной точки, опять запустить транслятор.

Одним из таких мест и является инструкция `include`. Как только управление программы доходит до нее, PHP вынужден приостановиться и ждать, пока транслятор не оттранслирует код включаемого файла. А это достаточно отрицательно сказывается на быстродействии программы. Поэтому, если вы пишете большой и сложный сценарий, применяйте инструкцию `require` вместо `include`, где только можно.

При использовании `include` PHP не сможет определить во время компиляции, какие файлы нужно подключить в программу, поэтому в исполняемый файл их код не войдет. Если определенный файл нужно присоединить ровно один раз и в точно определенное место, то лучше воспользоваться `require`, в противном случае более удачным выбором будет `include`.

Инструкции однократного включения. В больших и непростых сценариях инструкции `include` и `require` применяются очень и очень часто. Поэтому становится довольно сложно контролировать, как бы случайно не включить один и тот же файл несколько раз. Для этого используются инструкции `include_once` и `require_once`.

Инструкция `require_once` работает точно так же, как и require, но за одним важным исключением. Если она видит, что затребованный файл уже был ранее включен, то она ничего не делает. Такой метод работы требует от PHP хранения полных имен всех подсоединенных файлов где-то в недрах интерпретатора. Инструкция `include_once` работает совершенно аналогично, но включает файл во время исполнения программы, а не во время трансляции.

Замечание. В PHP существует внутренняя таблица, которая хранит полные имена всех включенных файлов. Проверка этой таблицы осуществляется инструкциями `include_once` и `require_once`. Однако *добавление* имени включенного файла производят также и функции `require` и `include`. Поэтому, если какой-то файл был востребован, например, по команде `require`, а затем делается попытка подключить его же, но с использованием `require_once`, то последняя инструкция просто проигнорируется.

Тема 5 Ассоциативные массивы

5.1 Списки и ассоциативные массивы
5.2 Операции над массивами
5.3 Списки и строки
5.4 Сериализация

5.1 Списки и ассоциативные массивы

Особенности ассоциативных массивов:

1. Все массивы PHP являются ассоциативными (в частности, списки – тоже).

2. Ассоциативные массивы в PHP являются направленными, т.е. в них существует определенный (и предсказуемый) порядок элементов, не зависящий от реализации, а значит, есть первый и последний элементы, и для каждого элемента можно определить следующий за ним.

3. Операция [] всегда добавляет элемент в конец массива, присваивая ему при этом такой числовой индекс, который бы не конфликтовал с уже имеющимися массиве (выбирается номер, превосходящий все имеющиеся цифровые ключи в массиве).

4. Операция `$Array[ключ]=значение` добавляет элемент в конец массива, за исключением тех случаев, когда ключ уже присутствует в массиве.

5. Для изменения порядка следования элементов в ассоциативном массиве, не изменяя их ключей, можно воспользоваться функциями сортировки, или создать новый пустой массив и заполнить его в нужном порядке, пройдясь по элементам исходного массива.

5.2 Операции над массивами

Над массивами можно выполнять следующие операции:

Инициализация массива.

```
$NamesList[0]="Dmitry";
$NamesList[1]="Helen";
$NamesList [2]="Sergey";
```

Печать элементов массива. Количество элементов в массиве легко можно определить функцией `count()` или ее синонимом `sizeof()`:

```
for($i=0; $i<count($NamesList); $i++)
   echo $NamesList[$i]."<br>";
```

Добавление элементов в массив (создание массива «на лету»). Автомассивы. Для добавления элемента в конец массива можно не задавать номер элемента массива.

```
$NamesList[]="Dmitry";
```

```
$NamesList[]="Helen";
$NamesList[]="Sergey";
```

В этом случае PHP сам начнет (если переменная $NamesList не существует) нумерацию с нуля и каждый раз будет прибавлять к счетчику по единичке, создавая список.

Ассоциативный массив часто называется просто ***хэш***.

Добавление элементов в ассоциативный массив осуществляется аналогично, только вместо цифровых ключей нужно указывать строковые. При этом следует помнить, что в строковых ключах буквы нижнего и верхнего регистров считаются различными. Ключом может быть абсолютно любая строка, содержащая пробелы, символы перевода строки, нулевые символы и т.д., т.е. никаких ограничений на ключи не накладывается.

Пример. Пусть необходимо написать сценарий, который работает, как записная книжка: по фамилии абонента выдает его имя. Можно организовать базу данных этой книжки в виде ассоциативного массива с ключами – фамилиями и соответствующими им значениями имен людей:

```
$Names["Koteroff"]="Dmitry";
$Names["Ivanov"]="Ivan";
$Names["Petrov"]="Peter";
```

Имя любого абонента можно распечатать командой:

```
echo $Names["Ivanov"] ;
$f="Koteroff";
echo $Names[$f];
```

Инструкция list(). Пусть есть некоторый массив-список $List с тремя элементами: имя человека, его фамилия и возраст. Нужно присвоить переменным $name, $surname и $age эти величины. Это можно сделать следующим образом:

```
$name=$List[0];
$surname=$List[1] ;
$age=$List[2];
```

Однако лучше воспользоваться инструкцией list():

```
list($name, $surname, $age)=$List;
```

Если в массиве не хватит элементов, чтобы их заполнить, им присвоятся неопределенные значения. Если нужны только второй и третий элемент массива $List, нужно пропустить первый параметр в инструкции list():

```
list(,$surname,$age)=$List;
```

Можно пропускать любое число элементов, как слева или справа, так и посередине списка. Главное – не забыть проставить нужное количество запятых.

Инструкция array() и многомерные массивы. Пусть необходимо написать программу, которая по фамилии некоторого человека из группы будет выдавать его имя. Данные хранятся в ассоциативном массиве:

```
$Names["Ivanov"]="Dmitry";
$Names ["Petrova"]="Helen";
```

Теперь можно написать:

```
echo $Names["Petrova"];   // выведет Helen
echo $Names["Oshibkov"]; // ошибка: в массиве нет такого
                                  // элемента!
```

То есть, таким образом нельзя создать пустой массив.

Второй способ создания массивов – это использование оператора array().

```
//создает пустой массив $Names
$Names=array();
//создает такой же массив, как в предыдущем примере с именами
$Names=array ("Ivanov"=>"Dmitry",
              "Petrova"=>"Helen")
//создает список с именами (нумерация 0,1,2)
$NamesList=array("Dmitry","Helen","Sergey");
```

Создание двумерных массивов. Если кроме имени о человеке известен также его возраст, то можно инициализировать массив $Names следующим образом:

```
$Names["Ivanov"]=array("name"=>"Dmitry","age"=>25);
$Names["Petrova"]=array("name"=>"Helen","age"=>23);
```

или так:

```
$Names=array(
"Ivanov"=>array("name"=>"Dmitry","age"=>25) ,
"Petrova"=>array("name"=>"Helen","age"=>23));
```

Получить нужный элемента массиве можно следующим образом:

```
echo $Names["Ivanov"]["age"]; // напечатает "25"
echo $Names["Petrova"]["bad"]; // ошибка: нет такого
                                        // элемента "bad"
```

Ассоциативные массивы в PHP удобно использовать как некие структуры, хранящие данные. Это похоже на конструкцию struct в Си (или record в Паскале). Это единственный возможный способ организации структур, но он очень гибок.

Доступ по ключу.

```
//выводит элемент массива $Arr с ключом anykey
echo $Arr ["anykey"];
//так используются двумерные массивы
echo $Arr ["first"]["second"];
```

```
echo $Arr[5];
```

Величина $Arr[ключ] является полноценным «левым значением», т. е. может стоять в левой части оператора присваивания, от нее можно брать ссылку с помощью оператора &, например:

```
//присваиваем элементу массива 100
$Arr["anykey"]=array(100,200);
//$ref – синоним элемента массива
$ref=&$Arr["first"]["second"];
$Arr[]="for add"; // добавляем новый элемент
```

***Функция* `count()`** – определяет число элементов массива:

```
$num=count($Names); // $num – число элементов в массиве
```

Слияние массивов, т.е. создание массива, содержащего как элементы одного, так и другого массива. Реализуется при помощи оператора +. Например:

```
$a=array("a"=>"aa", "b"=>"bb") ;
$b=array("c"=>"cc", "d"=>"dd");
$c=$a+$b;
```

В результате в $c окажется ассоциативный массив, содержащий все 4 элемента:

```
array("a"=>"aa","b"=>"bb","c"=>"cc","d"=>"dd"),
```

причем именно в указанном порядке.

Направленность массивов заставляет оператор + стать некоммутативным, т. е. $a+$b не равно $b+$a, если $a и $b массивы.

Так как списки являются тоже ассоциативными массивами, оператор + будет работать с ними неправильно! При слиянии списков:

```
$a=array(10,20,30);
$b=array(100,200);
$c=$a+$b;
```

в $c будет `array(10,20,30)` !

Потому что при конкатенации массивов с одинаковыми элементами (то есть, элементами с одинаковыми ключами) в результирующем массиве останется только один элемент с таким же ключом – тот, который был в первом массиве, и на том же самом месте.

При слиянии ассоциативных массивов

```
$a=array("a"=>10, "b"=>20);
$b=array("b"=>"new?");
$a+=$b;
```

оператор += ***не обновит*** элементы $a при помощи элементов $b. В результате этих операций значение $a не изменится.

Обновить элементы в массиве $a можно с помощью цикла:

```
foreach ($b as $k=>$v) $a[$k]=$v;
```

Цепочка


```
$z=$a+$b+$c+ ...
```

эквивалентна

```
$z=$a; $z+=$b; $z+=$c; ...и т. д.
```

Оператор += для массивов добавляет в свой левый операнд элементы, перечисленные в правом операнде-массиве, *если они еще не содержатся* в массиве слева.

В массиве не может быть двух элементов с одинаковыми ключами, потому что все операции, применимые к массивам, всегда контролируют, чтобы этого не произошло!

Косвенный перебор элементов массива. Используется для перебора всех элементов массива.

Если массив – список:

```
//Пусть $Names – список имен. Распечатаем их в столбик
for($i=0; $i<count ($Names); $i++)
echo $Names[$i]."\n";
```

Переменную можно помещать в строку:

```
for ($i=0; $i<count($Names); $i++)
echo "$Names[$i]\n";
```

Можно помещать массивы в строки, заключив их в фигурные скобки вместе с символом $:

```
$Names=array(array('name'=>'Вася','age'=>20),
             array('name'=>'Билл','age'=>40));
for ($i=0; $i<count($Names); $i++)
echo "{$Names[$i]['age']}\n";
```

Пусть массив $Names – ассоциативный: его ключи – имена людей, а значения, сопоставленные ключам – возраст этих людей. Для перебора такого массива можно воспользоваться следующей конструкцией:

```
for(Reset($Names);($k=key($Names));Next($Names))
   echo "Возраст $k - {$Names[$k]} лет\n";
```

Кроме того, что массивы являются направленными, в них есть еще и такое понятие, как текущий элемент. Функция Reset() устанавливает этот элемент на первую позицию в массиве. Функция key() возвращает ключ, который имеет текущий элемент (если он указывает на конец массива, возвращается пустая строка, что позволяет использовать вызов key() в контексте второго выражения for), а функция Next() перемещает текущий элемент на одну позицию вперед.

Функции Reset() и Next() возвращают следующие значения:

– функция Reset() возвращает значение первого элемента массива (или пустую строку, если массив пуст);

– функция Next() возвращает значение элемента, следующего за текущим (или пустую строку, если такого элемента нет).

Если необходим перебор элементов массива с конца, то можно воспользоваться следующей конструкцией:

```
for(End($Names);($k=key($Names));Prev($Names))
   echo "Возраст $k – {$Names[$k]} лет\n";
```

Функция End() устанавливает позицию текущего элемента в конец массива, а Prev() передвигает ее на один элемент назад.

Функция current() возвращает не ключ, а величину текущего элемента (если он не указывает на конец массива).

Основное *достоинство* косвенного перебора – «читабельность» и ясность кода, а также то, что массив можно перебрать как в одну, так и в другую сторону.

Недостатки косвенного перебора:

1 Вложенные циклы. Нельзя одновременно перебирать массив в двух вложенных циклах или функциях. Так как второй вложенный for «испортит» положение текущего элемента у первого for'а.

2 Нулевой ключ. Если в массиве встретится ключ 0:

```
for(Reset($Names);($k=key($Names));Next($Names))
   echo "Возраст $k – {$Names[$k]} лет\n";
```

то в этом случае выражение ($k=key ($Names)), будет равно нулю и цикл оборвется.

Прямой перебор массива. В косвенном переборе сначала вычисляется очередной ключ, а затем по нему косвенно находится значение элемента массива, при прямом переборе на каждом «витке» цикла одновременно получается и ключ, и значение текущего элемента.

Классический перебор. Пусть массив $Names хранит связь имен людей и их возрастов. Перебрать этот массив при помощи прямого перебора можно следующим образом:

```
for(Reset($Names);list($k,$v)=each($Names);
    /*пусто*/)
   echo "Возраст $k – $v\n";
```

В начале заголовка цикла используется функция Reset(). Дальше переменным $k и $v присваивается результат работы функции each(). Третье условие цикла просто отсутствует.

Функция each() во-первых, возвращает список, нулевой элемент которого хранит величину ключа текущего элемента массива $Names, а первый – значение текущего элемента. Во-вторых, она продвигает указатель текущего элемента к следующей позиции. Если следующего элемента в массиве нет, то функция возвращает не список, а false. Поэтому она и размещена в условии цикла for и не указан третий блок операторов в цикле for: он просто не нужен, ведь указатель на текущий элемент и так смещается функцией each ().

Перебор в стиле PHP. Прямой перебор массивов можно осуществить с помощью функции foreach:

```
foreach($Names as $k=>$v)
   echo "Возраст $k – $v\n";
```

Этот способ перебора работает с максимально возможной скоростью – даже быстрее, чем перебор списка при помощи for и числового счетчика.

5.3 Списки и строки

Функция explode() используется для разбиения строки на более мелкие части (эти части разделяются в строке каким-то специфическим символом типа |). Формат задания функции:

```
list explode (string $token, string $Str
              [, int $limit])
```

Она получает строку, заданную в ее втором аргументе, и пытается найти в ней подстроки, равные первому аргументу. Затем по месту вхождения этих подстрок строка «разрезается» на части, помещаемые в массив-список, который и возвращается. Если задан параметр $limit, то учитываются только первые ($limit-1) участков «разреза». Таким образом, возвращается список из не более чем $limit элементов.

```
$st="4597219361|Иванов|Иван|40|ivan@ivanov.com|Текс
т, содержащий (|)";
$A=explode("|", $st, 6);
// Мы знаем, что там только 6 полей!
//Теперь $A[0] ="Иванов", ... $A[5]= "Текст, содержащий (|)!"
// распределили по переменным
list ($Surname,$Name,$Age,$Email,$Tel)=$A;
```

Строкой разбиения может быть не только один символ, но и небольшая строка.

Функция implode()и ее синоним join() используются для слияния нескольких небольших строк в одну большую, вставляя между ними разделитель:

```
string implode (string $glue, list $List)
```

или

```
string join(string $glue, list $List).
```

Они берут ассоциативный массив (обычно это список) $List, заданный в ее первом параметре, и «склеивают» его значения при помощи «строки-клея» $glue во втором параметре.

Вместо списка во втором аргументе можно передавать любой ассоциативный массив – в этом случае будут рассматриваться только его значения.

5.4 Сериализация

С помощью функций `implode()` и `explode()` можно просто сохранить массив, например, в файле, а затем его оттуда считать и быстро восстановить. Однако этот способ имеет ряд существенных недостатков: во-первых, таким образом можно сохранять только массивы-списки (потому что ключи в любом случае теряются), а во-вторых, ничего не выйдет с многомерными массивами. Пусть нам все-таки нужно сохранить какой-то массив (причем неизвестно заранее, сколько у него измерений) в файле, чтобы потом, при следующем запуске сценария, его аккуратно загрузить и продолжить работу.

Для упаковки массива в строку (ведь в файлы можно писать только строки), и для восстановления строки в массив в исходном виде используются функции `Serialize()` и `Unserialize()`.

Функция `Serialize()` возвращает строку, являющуюся упакованным эквивалентом некоего объекта `$obj`, переданного во втором параметре.

```
string Serialize (mixed $0bj )
```

При этом совершенно не важно, что это за объект: массив, целое число.

Пример:

```
$A=array ("a"=>"aa", "b"=>"bb",
          "c" = >array ("x"=>"xx"));
$st=Serialize($A) ;
echo $st;
// выведется что-то типа нечто:
//a:2:(s:1:"a";s:2:"aa";s:1:"b";s:2:"bb";s:1:"c";a:1:(s:1:"x";s:2:"xx";))
```

Функция `Unserialize()`, наоборот, принимает в лице своего параметра `$st` строку, ранее созданную при помощи `Serialize()`, и возвращает целиком объект, который был упакован.

```
mixed Unserialize (string $st)
```

Например:

```
$a=array(l,2,3);
$s=Serialize($a);
$a= "bogus";
echo count ($a) ;   //     выводит 1
$a=Unserialize ($s) ;
echo count($a);     //     выводит 3
```

Сериализовать можно не только массивы, но и вообще что угодно. Однако в большинстве случаев все-таки используются массивы. Механизм сериализации часто применяется также и для того, чтобы сохранить какой-то объект в базе данных, и тогда без сериализации практически не обойтись.

Тема 6 Работа с массивами

6.1 Сортировка массивов
6.2 Функции для работы с массивами

6.1 Сортировка массивов

Сортировка массива по значениям (`asort()/arsort()`).

Функция asort() сортирует массив, указанный в ее параметре в алфавитном (если это строки) или в возрастающем (для чисел) порядке. При этом сохраняются связи между ключами и соответствующими им значениями. Пример:

```
$A=array("a"=>"Zero","b"=>"Weapon","c"=>"Alpha",
         "d"=>"Processor");
asort($A);
foreach($A as $k=>$v) echo "$k=>$v ";
// выводит «c=>Alpha d=>Processor b=>Weapon a=>Zero»
```

Функция `arsort()` упорядочивает массив по убыванию.

Сортировка по ключам (`ksort()/krsort()`).

Функция `ksort()` сортирует массив по ключам (в порядке возрастания). Пример:

```
$A=array("d"=>"Zero","c"=>"Weapon","b"=>"Alpha","a"
=>"Processor");
ksort($A);
for (Reset($A); list ($k, $v) =each ($A);)
echo"$k=>$v";
// выводит "a=>Processor b=>Alpha c=>Weapon d=>Zero"
```

Функция для сортировки по ключам в обратном порядке – `krsort()`.

Сортировка по ключам при помощи функции `uksort()`.

Используется при сортировке по более сложному критерию, чем просто по алфавиту. Например, пусть в $Files хранится список имен файлов и подкаталогов в текущем каталоге. Необходимо вывести этот список не только в лексикографическом порядке, но также и чтобы все каталоги предшествовали файлам. В этом случае можно воспользоваться функцией `uksort()`, написав предварительно функцию сравнения с двумя параметрами.

Пример. Сортировка с помощью пользовательской функции

```
// Эта функция должна сравнивать значения $f1 и $f2 и возвращать:
// -1, если $fl<$f2,
// 0, если $fl==$f2
// 1, если $fl>$f2
function FCmp($f1,$f2)
```

```
{
// Каталог всегда предшествует файлу
if (is_dir($f1) && !is_dir($f2)) return -1;
// Файл всегда идет после каталога
if (!is_dir($f1) && is_dir($f2)) return 1;
// Иначе сравниваем лексикографически
if($f1<$f2) return -1; elseif ($f1>$f2) return 1;
else return 0;
}
// Пусть $Files содержит массив с ключами — именами файлов
// в текущем каталоге. Отсортируем его.
uksort($Files,"FCmp");
// передаем функцию сортировки "по ссылке"
```

Связи между ключами и значениями функцией `uksort()` сохраняются.

Сортировка по значениям при помощи функции `uasort()`.

Функция `uasort()` аналогична `uksort()`, с той разницей, что пользовательской функции сортировки передаются не ключи, а очередные значения из массива. При этом также сохраняются связи в парах ключ=> значение.

Переворачивание массива `array_reverce()`

Функция `array_reverse()` возвращает массив, элементы которого следуют в обратном порядке относительно массива, переданного в параметре. При этом связи между ключами и значениями сохраняются. Например, вместо того, чтобы ранжировать массив в обратном порядке при помощи `arsort()`, можно отсортировать его в прямом порядке, а затем перевернуть:

```
$A=array ("a"=>"Zero", "b"=>"Weapon", "c"=>"Alpha",
          "d"=>"Processor");
asort ($A);
$A=array_reverse($A);
```

Сортировка списка `sort()/rsort()`

Эти две функции предназначены для сортировки списков (под списками понимаются массивы, ключи которых начинаются с 0 и не имеют пропусков). Функция `sort()` сортирует список по значениям в порядке возрастания, а `rsort()` – в порядке убывания. Пример:

```
$A=array ("One", "Two", "Three", "Four");
sort ($A) ;
for($i=0; $i<count($A); $i++) echo "$i:$A[$i] ";
// выводит «0: Four 1: Two 2 : Three 3 :One»
```

Любой ассоциативный массив воспринимается этими функциями как список. То есть после упорядочивания последовательность ключей

превращается в 0,1,2,..., а значения нужным образом перераспределяются. Связи между парами ключ=>значение не сохраняются, более того – ключи просто пропадают, поэтому сортировать что-либо, отличное от списка, нецелесообразно.

Сортировка списка при помощи функции `usort()`.

Эта функция является «гибридом» функций `uasort()` и `sort()`. От `sort()` она отличается тем, что критерий сравнения обеспечивается пользовательской функцией. А от `uasort()` – тем, что она не сохраняет связей между ключами и значениями, а потому пригодна только для сортировки списков. Пример:

```
function FCmp($a,$b)
{ return strcmp ($a, $b); }
$A=array("One", "Two", "Three", "Four");
usort($A);
for ($i=0; $i<count ($A); $i++) echo "$i:$A[$i]";
// выводит «0:Four 1:One 2:Three 3:Two»
```

Функция `strcmp()`, возвращает –1, если `$a<$b`, 0, если они равны, и 1, если `$a>$b`. Приведенный здесь пример полностью эквивалентен простому вызову `sort()`.

Перемешивание списка `shuffle()`.

Функция `shuffle()` «перемешивает» список, переданный ей первым параметром, так, чтобы его значения распределялись случайным образом. При этом, во-первых, изменяется сам массив, а во-вторых, ассоциативные массивы воспринимаются как списки. Пример:

```
$A=array(10,20,30,40,50);
shuffle($A);
foreach($A as $v) echo "$v ";
```

Приведенный фрагмент выводит числа 10, 20, 30, 40 и 50 в случайном порядке. Функция `shuffle()` использует стандартный генератор случайных чисел, который перед работой необходимо инициализировать при помощи вызова `srand()`, иначе при повторном вызове программы результат не измениться.

6.2 Функции для работы с массивами

Ключи и значения.

- `array array_flip(array $Arr)`

Функция меняет местами его ключи и значения. Исходный массив `$Arr` не изменяется, а результирующий массив просто возвращается. Если в массиве присутствовали несколько элементов с одинаковыми значениями, учитываться будет только последний из них:

```
$A=array("a"=>"aaa", "b"=>"aaa", "c"=>"ccc");
$A=array_flip($A);
```

```
// теперь $A===array("aaa"=>"b", "ccc"=>"c");
```

- `list array_keys(array $Arr [,mixed $SearchVal])`

Функция возвращает список, содержащий все ключи массива `$Arr`. Если задан необязательный параметр `$SearchVal`, то она вернет только те ключи, которым соответствуют значения `$SearchVal`.

Фактически, эта функция с заданным вторым параметром является обратной по отношению к оператору [] — извлечению значения по его ключу.

- `list array_values(array $Arr)`

Функция `array_values()` возвращает список всех значений в ассоциативном массиве `$Arr`. Такое действие бесполезно для списков, но иногда оправдано для хэшей.

- `bool in_array(mixed $val, array $Arr)`

Возвращает `true`, если элемент со значением `$val` присутствует в массиве `$Arr`.

- `array array_count_values(list $List)`

Функция подсчитывает, сколько раз каждое значение встречается в списке `$List`, и возвращает ассоциативный массив с ключами – элементами списка и значениями – количеством повторов этих элементов. Т.е. функция `array_count_values()` подсчитывает частоту появления значений в списке `$List`. Пример:

```
$List=array(1, "hello", 1, "world", "hello");
array_count_values($array);
// возвращает array(1=>2, "hello"=>2, "world"=>1)
```

Комплексная замена в строке.

`string strtr(string $st, array $Substitutes)`

Функция `strtr` берет строку `$st` и проводит в ней контекстный поиск и замену: ищутся подстроки – ключи в массиве `$Substitutes` – и замещаются на соответствующие им значения.

```
$Subs=array("<name>" => "Larry",
            "<time>" => date("d.m.Y"));
$st="Привет, <name>! Сейчас <time>";
echo strtr($st,$Subs);
```

Функция `strtr()` начинает поиск с самой длинной подстроки и не проходит по одному и тому же ключу дважды.

Слияние массивов.

`array array_merge(array $A1, array $A2, ...)`

Функция `array_merge()` сливает массивы, перечисленные в ее аргументах, в один большой массив и возвращает результат. Если в массивах встречаются одинаковые ключи, в результат помещается пара ключ=>значение из того массива, который расположен правее в списке

аргументов. Однако это не затрагивает числовые ключи: элементы с такими ключами помещаются в конец результирующего массива в любом случае. Пример, сливающий два списка в один:

```
$L1=array(10,20,30);
$L2=array(100,200,300);
$L=array_merge($L1,$L2);
// теперь $L===array(10,20,30,100,200,300);
```

Получение части массива.

`array array_slice(array $Arr, int $offset [, int $len])`

Функция возвращает часть ассоциативного массива, начиная с пары ключ=>значения со смещением (номером) `$offset` от начала и длиной $len (если последний параметр не задан, до конца массива). Параметры `$offset` и `$len` задаются по точно таким же правилам, как и аналогичные параметры в функции `substr()`. Параметры могут быть отрицательными (в этом случае отсчет осуществляется от конца массива), и т. д. Примеры:

```
$input=array("a","b","c","d","e");
$output=array_slice($input,2);      // "c", "d", "e"
$output=array_slice($input,2,-1);   // "c", "d"
$output=array_slice($input,-2,1);   // "d"
$output=array_slice($input,0,3);    // "a", "b", "c"
```

Вставка/удаление элементов.

Для вставки и удаления элементов массива могут использоваться: оператор `[]` (пустые квадратные скобки) добавляет элемент в конец массива, присваивая ему числовой ключ; оператор `Unset()` вместе с извлечением по ключу удаляет нужный элемент.

- `int array_push(alist &$Arr, mixed $var1 [, mixed $var2, …])`

Функция добавляет к списку `$Arr` элементы `$var1`, `$var2` и т. д. Она присваивает им числовые индексы – точно так же, как это происходит для стандарных `[]`.

```
array_push($Arr,1000);  // вызываем функцию…
$Arr[]=1000;            // то же самое, но короче
```

Функция `array_push()` воспринимает массив, как стек, и добавляет элементы всегда в его конец. Она возвращает новое число элементов в массиве.

- `mixed array_pop(list &$Arr)`

Функция `array_pop()`, в противоположность `array_push()`, снимает элемент с «вершины» стека (то есть берет последний элемент

списка) и возвращает его, удалив после этого его из $Arr. Если список $Arr был пуст, функция возвращает пустую строку.

- `int array_unshift(list &$Arr, mixed $var1 [,mixed $var2, …])`

Функция похожа на array_push(), но добавляет перечисленные элементы не в конец, а в начало массива. При этом порядок следования $var1, $var2 и т. д. остается тем же, т. е. элементы как бы «вдвигаются» в список слева. Новым элементам списка, как обычно, назначаются числовые индексы, начиная с 0; при этом все ключи старых элементов массива, которые также были числовыми, изменяются (чаще всего они увеличиваются на число вставляемых значений). Функция возвращает новый размер массива. Пример:

```
$A=array(10,"a"=>20,30);
array_unshift($A,"!","?");
// теперь $A===array(0=>"!", 1=>"?", 2=>10, a=>20, 3=>30)
```

- `mixed array_shift(list &$Arr)`

Функция извлекает первый элемент массива $Arr и возвращает его. При извлечении первого элемента корректируются все числовые индексы у всех оставшихся элементов.

- `array array_unique(array $Arr)`

Функция array_unique() возвращает массив, составленный из всех уникальных значений массива $Arr вместе с их ключами. В результирующий массив помещаются первые встретившиеся пары ключ=>значение:

```
$input=array("a" => "green", "red", "b" => "green",
"blue", "red");
$result=array_unique($input);
// теперь $result===array("a"=>"green", "red", "blue");
```

- `array array_splice(array &$Arr, int $offset [, int $len] [, int $Repl])`

Функция возвращает подмассив $Arr, начиная с индекса $offset максимальной длины $len, и заменяет только что указанные элементы на то, что находится в массиве $Repl (или просто удаляет, если $Repl не указан). Параметры $offset и $len могут быть и отрицательными, в этом случае отсчет начинается от конца массива. Пример:

```
$input=array("red", "green", "blue", "yellow");
array_splice($input,2);
// Теперь $input===array("red", "green")
array_splice($input,1,-1);
// Теперь $input===array("red", "yellow")
```



```
array_splice($input,-1,1,array("black","maroon"));
// Теперь $input===array("red", "green", "blue", "black", "maroon")
array_splice($input,1,count($input),"orange");
// Теперь $input===array("red", "orange")
```

В качестве параметра $Repl можно указать и обычное, строковое значение, а не массив из одного элемента.

Переменные и массивы.

- `array compact(mixed $vn1 [, mixed $vn2, …])`

Функция compact(), упаковывает в массив переменные из текущего контекста (глобального или контекста функции), заданные своими именами в $vn1, $vn2 и т. д. При этом в массиве образуются пары с ключами, равными содержимому $vnN, и значениями соответствующих переменных. Пример:

```
$a="Test string";
$b="Some text";
$A=compact("a","b");
// теперь $A===array("a"=>"Test string", "b"=>"Some text")
```

Параметры функции могут быть не только строками, но и списками строк. В этом случае функция последовательно перебирает все элементы этого списка, и упаковывает те переменные из текущего контекста, имена которых она встретила. Эти списки могут, в свою очередь, также содержать списки строк, и т. д. Пример:

```
$a="Test";
$b="Text";
$c="CCC";
$d="DDD";
$Lst=array("b",array("c","d"));
$A=compact("a",$Lst);
//теперь $A===array("a"=>"Test","b"=>"Text","c"=>"CCC", "d"=>"DDD")
```

- `void extract(array $Arr [, int $type] [, string $prefix])`

Функция получает в параметрах массив $Arr и превращает каждую его пару ключ=>значение в переменную текущего контекста. Параметр $type предписывает, что делать, если в текущем контексте уже существует переменная с таким же именем, как очередной ключ в $Arr. Он может быть равен одной из следующих констант:

EXTR_OVERWRITE – переписывать существующую переменную (по умолчанию);

EXTR_SKIP – не перезаписывать переменную, если она уже существует;

EXTR_PREFIX_SAME – в случае совпадения имен создавать переменную с именем, предваренным префиксом из $prefix.

EXTR_PREFIX_ALL – всегда предварять имена создаваемых переменных префиксом $prefix

Пример:

```
// Сделать все переменные окружения глобальными
extract($HTTP_ENV_VARS);
// То же самое, но с префиксом E_
extract($HTTP_ENV_VARS, EXTR_PREFIX_ALL, "E_");
echo $E_COMSPEC; // выводит переменную окружения COMSPEC
```

Параметр $prefix имеет смысл указывать только тогда, применяются режимы EXTR_PREFIX_SAME или EXTR_PREFIX_ALL.

Замечание. Использование extract() и compact() может быть оправдано лишь для небольших массивов, да и то только в шаблонах, а в остальных случаях считается признаком дурного тона.

Создание списка – диапазона чисел.

```
list range(int $low, int $high)
```

Функция создает список, заполненный целыми числами от $low до $high включительно.

Тема 7 Функции и области видимости

7.1 Формат определения функции, передача параметров
7.2 Глобальные, локальные, статические переменные
7.3 Рекурсия, вложенные и условно-определяемые функции
7.4 Передача функций по ссылке и возврат функцией ссылки

7.1 Формат определения функции, передача параметров

Формат определения функции:

```
function имя_функции (арг1[=зн1], арг2[=зн2], ... аргN[=знN])
{
операторы_тела_функции;
}
```

Имя функции должно быть уникальным с точностью до регистра букв. Это означает, что, во-первых, имена MyFunction, myfunction и даже MyFuNcTiOn будут считаться одинаковыми, и, во-вторых, нельзя переопределить уже определенную функцию (стандартную или нет – не важно), но зато можем давать функциям такие же имена, как и переменным в программе (конечно, без знака $ в начале). Список аргументов состоит из нескольких перечисленных через запятую переменных, каждая из которых должна задаваться при вызове функции (если для этой переменной присвоено через знак равенства значение по умолчанию (обозначенное =знМ), ее можно будет опустить).

Если у функции не должно быть аргументов, то следует оставить пустые скобки после ее имени, например:

```
function SimpleFunction()
{ ... }
```

В фигурные скобки заключается *тело функции*. В нем могут быть любые операторы, включая даже операторы определения других функций (однако эти «другие функции» не будут локальными, как в Паскале, а станут далее «видны» для всей программы, но только с того момента, как до их описания дойдет управление). Если функция должна возвращать какое-то значение, что среди них должен встретиться оператор return. Если же она должна отработать без возврата значений, то оператор return можно и не указывать (или указывать без задания возвращаемого значения).

Синтаксис оператора return:

```
return возвращаемое значение;
```

В PHP можно использовать return абсолютно для любых объектов (какими бы большими они ни были), причем без заметной потери быстродействия. Пример:

```
function MySqrt($n)
{ return $n*$n;
}
echo MySqrt(4); // выводит 16
```

Сразу несколько значений функции возвратить не могут. Однако, если это все же нужно, то можно вернуть ассоциативный массив или список:

```
function Silly()
{ return array(1,2,3);
}
// присваивает массиву значение array(1,2,3)
$arr=Silly();
// присваивает переменным $a, $b, $c первые значения из списка
list($a,$b,$c)=Silly();
```

Если функция не возвращает никакого значения, т. е. инструкции `return` в ней нет, то считается, что функция возвратила значение `false`.

Вызывать функции нужно только после того, как они будут определены.

В отличие от других языков программирования, функцию можно задавать не только в определенном месте программы, но и прямо среди других операторов:

```
echo "Программа...";
function GetMaxNum($arr,$max)
{ ... тело функции ...
}
echo "Программа продолжается!";
```

При таком подходе транслятор, дойдя до определения функции, проверит его корректность и оттранслирует во внутреннее представление, но не будет генерировать код для выполнения, а сразу переключится на следующие за телом функции команды. Только потом, при вызове функции, интерпретатор начнет исполнять ее команды.

Параметры по умолчанию. Часто бывают необходимо, чтобы у функции было много параметров, причем некоторые из них будут задаваться совершенно единообразно. Например, нужна функция сортировки массива. Тогда, кроме очевидного параметра – массива – хотелось бы также задавать и второй параметр, который бы указывал порядок сортировки (в убывающем или в возрастающем порядке). При этом чаще всего придется сортировать в порядке убывания. В этом случае мы можем оформить функцию так:

```
function MySort(&$Arr, $NeedLoOrder=1)
{ ... сортируем в зависимости от $NeedLoOrder...
}
```


Теперь, имея такую функцию, можно написать в программе:

```
MySort($my_array,0); // сортирует в порядке возрастания
MySort($my_array); // второй аргумент задается по умолчанию!
```

То есть, можно опустить второй параметр у функции, что будет выглядеть так, как будто его задали равным 1. Значение по умолчанию для какого-то аргумента указывается справа от него через знак равенства. Значения аргументов по умолчанию должны определяться справа налево, причем недопустимо, чтобы после любого из таких аргументов шел обычный «неумолчальный» аргумент. Пример неверного описания функции:

```
// Ошибка!
function MySort($NeedLoOrder=1, &$Arr)
{
... сортируем в зависимостиот $NeedLoOrder...
}
MySort(,$my_array); // Ошибка!
```

Передача параметров по ссылке. Рассмотрим механизм передачи аргументов функции. Пусть, например, есть такая программа:

```
function Test($a)
{ echo "$a\n";     // распечатается значение 10
$a++;
echo "$a\n";       // распечатается значение 11
}
. . .
$num=10;
Test($num);
echo $num;      // распечатается значение 10
```

Функция `Test()` не возвращает никакого значения, т. е. является в чистом виде подпрограммой или процедурой. Создается переменная `$a`, *локальная* для данной функции, и ей присваивается значение 10 (то, что было в `$num`). После этого значение 10 выводится на экран, величина `$a` инкрементируется, и новое значение (11) опять печатается. Так как тело функции закончилось, происходит возврат в вызвавшую программу. При последующем выводе переменной `$num` будет напечатано 10 (и это несмотря на то, что в переменной `$a` до возврата из функции было 11!). Это происходит потому, что `$a` – лишь *копия* `$num`, а изменение копии никак не отражается на оригинале.

Если необходимо, чтобы функция имела доступ не к величине, а именно к *самой переменной* (переданной ей в параметрах), достаточно при передаче аргумента функции перед его именем поставить `&`.

Пример передачи параметров по ссылке (первый способ):

```
function Test($a)
```

```
{ echo "$a\n";
$a++;
echo "$a\n";
}
$num=10;            // $num=10
Test(&$num);        // теперь $num=11!
echo $num;          // выводит 11!
```

Такой способ передачи параметров исторически называется «передачей по ссылке», в этом случае аргумент не является копией переменной, а «ссылается» на нее. Передача параметра по ссылке полностью соответствует синтаксису задания ссылочной переменной в PHP.

Чтобы не забывать каждый раз писать `&` перед переменной, передавая ее функции, существует и другой синтаксис передачи по ссылке: можно символ `&` перенести прямо в заголовок функции.

Передача параметров по ссылке (второй способ):

```
function Test(&$a)
{ echo "$a\n";
$a++;
echo "$a\n";
}
....
$num=10;        // $num=10
Test($num);     // а теперь $num=11!
echo $num;      // выводит 11!
```

В качестве параметров, передаваемых по ссылке, можно задавать только переменные, но не непосредственные значения!

Переменное число параметров. Функция может иметь несколько параметров, заданных по умолчанию. Они перечисляются справа налево, и их всегда фиксированное количество. Пусть необходимо написать функцию, которая принимает один или более параметров (сколько именно – неизвестно на этапе определения функции). Пусть она должна вывести эти параметры «лесенкой» – каждый следующий на новой строке с отступом от предыдущего.

Пример передачи переменного числа параметров функции:

```
function myecho()
{ for($i=0; $i<func_num_args(); $i++)
{
for($j=0; $j<$i; $j++) echo " ";// выводим отступ
echo func_get_arg($i)."<br>\n"; // выводим элемент
}
}
// отображаем строки «лесенкой»
```



```
myecho("Меркурий", "Венера", "Земля", "Марс");
```

При описании `myecho()` указаны пустые скобки в качестве списка параметров, словно функция не получает ни одного параметра. На самом деле в PHP при вызове функции можно указывать параметров больше, чем задано в списке аргументов – в этом случае никакие предупреждения не выводятся (но если фактическое число параметров меньше, чем указано в описании, PHP выдаст сообщение об ошибке). «Лишние» параметры как бы игнорируются, в результате пустые скобки в `myecho()` позволяют в действительности передать ей сколько угодно параметров.

Для того чтобы иметь доступ к «проигнорированным» параметрам, существуют три встроенные в PHP функции:

- `int func_num_args()`

Возвращает *общее* число аргументов, переданных функции при вызове.

- `mixed func_get_arg(int $num)`

Возвращает значение аргумента с номером `$num`, заданного при вызове функции. Нумерация отсчитывается с нуля.

- `list func_get_args()`

Возвращает список всех аргументов, указанных при вызове функции.

Пример использования функции `func_get_args`:

```
function myecho()
{ foreach(func_get_args() as $v)
{
for($j=0; $j<@$i; $j++) echo " ";
echo "$v<br>\n";
@$i++;
}
}
// выводим строки "лесенкой"
myecho("Меркурий", "Венера", "Земля", "Марс");
```

В программе используется цикл `foreach` для перебора аргументов, а также оператор отключения ошибок `@`, чтобы PHP не «ругался» на то, что переменная `$i` не определена при первом «обороте» цикла.

7.2 Глобальные, локальные, статические переменные

Все переменные, которые объявляются и используются в функции, по умолчанию локальны для этой функции. При этом существует только два способа объявления глобальных переменных: инструкция `global` или через массив `$GLOBALS`.

Локальные переменные. Аргументы функции (передаваемые по значению, а не по ссылке) являются временными объектами, которые создаются в момент вызова и исчезают после окончания функции.

Пример локальных переменных (параметров):

```
$a=100;  // Глобальная переменная, равная 100
function Test($a)
{ echo $a;  // выводим значение параметра $a
// Этот параметр не имеет к глобальной $a никакого отношения!
$a++;    // изменяется только локальная копия значения,
         //переданного в $a
}
Test(1);  // выводит 1
echo $a;  // выводит 100 – глобальная $a не изменилась
```

Такими же свойствами будут обладать не только аргументы, но и все другие переменные, инициализируемые или используемые внутри функции.

Пример локальных переменных:

```
function Silly()
{ $i=rand();  // записывает в $i случайное число
echo $i;     // выводит его на экран
             // Эта $i не имеет к $i никакого отношения!
}
for($i=0; $i!=10; $i++) Silly();
```

Здесь переменная $i в функции будет не той переменной $i, которая используется в программе для организации цикла. Поэтому цикл и проработает только 10 «витков», напечатав 10 случайных чисел (а не будет крутиться долго и упорно, пока «в рулетке» функции rand() не выпадет 10.

Глобальные переменные. Для задания глобальной переменной нужно до её первого использования объявить ее в теле функции «глобальной».

Пример использование global:

```
function Silly()
{  global $i;
   $i=rand();
   echo $i;
}
for($i=0; $i!=10; $i++) Silly();
```

Теперь переменная $i будет везде едина: что в функции, что во внешнем цикле (для последнего это приведет к немедленному его «зацикливанию», во всяком случае до тех пор, пока rand() не выкинет 10).

Пример использования глобальных переменных внутри функции:

```
$Monthes[1]="Январь";
$Monthes[2]="Февраль";
...
$Monthes[12]="Декабрь";
. . .
// Возвращает название месяца по его номеру.
// Нумерация начинается с 1!
function GetMonthName($n)
{ global $Monthes;
  return $Monthes[$n];
}
. . .
echo GetMonthName(2);  // выводит «Февраль»
```

Массив $Monthes, содержащий названия месяцев, довольно объемист. Поэтому описывать его прямо в функции неудобно. В то же время функция GetMonthName() представляет собой довольно преемлемое средство для приведения номера месяца к его словесному эквиваленту. Она имеет единственный параметр: это номер месяца.

Массив $GLOBALS. Есть и второй способ добраться до глобальных переменных. Это – использование встроенного в язык массива $GLOBALS. Он представляет собой хэш, ключи которого есть имена глобальных переменных, а значения – их величины. Этот массив доступен из любого места в программе – в том числе и из тела функции, и его не нужно никак дополнительно объявлять.

Пример использования массива $GLOBAL:

```
// Возвращает название месяца по его номеру.
// Нумерация начинается с 1!
function GetMonthName($n)
{ return $GLOBALS["Monthes"][$n]; }
```

Не только переменные, но даже и массивы могут иметь совершенно любую структуру, какой бы сложной она ни была. Например, предположим, что в программе есть ассоциативный массив $A, элементы которого – двумерные массивы чисел. Тогда доступ к какой-нибудь ячейке этого массива с использованием $GLOBALS мог бы выглядеть так:

```
$GLOBALS["A"]["First"][10][20];
```

То есть получился четырехмерный массив!

Массив $GLOBALS имеет ряд особенностей.

Во-первых, массив $GLOBALS изначально является глобальным для любой функции, а также для самой программы. Вполне допустимо

его использовать не только в теле функции, но также и в любом другом месте.

Во-вторых, с этим массивом допустимы не все операции, разрешенные с обычными массивами. Нельзя присвоить этот массив какой-либо переменной целиком, используя оператор =; нельзя передать его функции «по значению» – можно передавать только по ссылке. Остальные операции допустимы. Можно по одному перебрать у массива $GLOBALS все элементы и, например, вывести их значения на экран.

В-третьих, добавление нового элемента в $GLOBALS равнозначно созданию новой глобальной переменной (предваренной символом $ в начале имени, ведь в самом массиве ключи – это имена переменных *без* символа доллара), а выполнение операции Unset() для него равносильно уничтожению соответствующей переменной.

В массиве $GLOBALS всегда присутствует переменная GLOBALS. Элемент с ключом GLOBALS является не обычным массивом, а лишь ссылкой на $GLOBALS.

Особенность использования инструкция `global`.

Инструкция `global $a` говорит о том, что переменная $a является глобальной, т.е., является синонимом глобальной $a. Синоним в терминах PHP – это ссылка, поэтому global создает ссылку.

Пример:

```
function Test()
{ global $a;
  $a=10;
}
```

Приведенное описание функции Test() полностью эквивалентно следующему описанию:

```
function Test()
{ $a=&$GLOBALS['a'];
  $a=10;
}
```

Из второго фрагмента видно, что оператор Unset($a) в теле функции не уничтожит глобальную переменную $a, а лишь «отвяжет» от нее ссылку $a. Точно то же самое происходит и в первом случае.

Пример:

```
$a=100;
function Test()
{ global $a;
Unset($a);
}
Test();
echo $a; // выводит 100, т. е. настоящая $a не была удалена в Test()!
```


Для того, чтобы удалить глобальную $a из функции существует только один способ: использовать $GLOBALS['a']:

```
function Test() { unset($GLOBALS['a']); }
$a=100;
Test();
echo $a; // Ошибка! Переменная $a не определена!
```

Статические переменные. Статические переменные определяются в теле функции с помощью слова static.

Пример задания статических переменных:

```
function Silly()
{ static $a=0;
  echo $a;
  $a++;
}
for($i=0; $i<10; $i++) Silly();
```

После запуска будет выведена строка 0123456789. Если убрать слово static, то результатом будет строка 0000000000, потому что переменная $a стала локальной, и ей при каждом вызове функции присваивается одно и то же значение – 0.

Конструкция static говорит компилятору о том, что уничтожать указанную переменную для функции между вызовами не надо. В то же время присваивание $a=0 сработает только один раз, а именно – при самом первом обращении к функции.

7.3 Рекурсия, вложенные и условно-определяемые функции

Рекурсия. В PHP поддерживаются рекурсивные вызовы функций, т. е. вызовы функцией самой себя.

Примера функции, рекурсивно вычисляющей факториал числа n.

```
function Factor($n)
{ if($n<=0) return 1;
  else return $n*Factor($n-1);
}
echo Factor(20);
```

Вложенные функции. Стандарт PHP не поддерживает вложенные функции. Однако он поддерживает нечто, немного похожее на них. Вместо того чтобы, как и у переменных, ограничить область видимости для вложенных функций своими «родителями», PHP делает их доступными для всей остальной части программы, но только с того момента, когда «функция-родитель» была из нее вызвана.

Пример вложенных функций:

```
function Parent($a)
{ echo $a;
```

```
  function Child($b)
  { echo $b+1;
     return $b*$b;
  }
  return $a*$a*Child($a);
  // фактически возвращает $a*$a*($a+1)*($a+1)
}
// Вызываем функции
Parent(10);
Child(30);
// Если теперь ВМЕСТО этих двух вызовов поставить такие
// же, но только в обратном порядке, то система выдает ошибку!
```

Нет никаких ограничений на место описания функции – будь то глобальная область видимости программы, либо же тело какой-то другой функции. В то же время понятия «локальная функция» как такового в PHP не существует. Каждая функция добавляется во внутреннюю таблицу функций PHP тогда, когда управление доходит до участка программы, содержащего определение этой функции. При этом, само тело функции пропускается, однако ее имя фиксируется и может далее быть использовано в сценарии для вызова. Если же в процессе выполнения программы PHP никогда не доходит до определения некоторой функции, она не будет «видна», как будто ее и не существует.

Если вызвать функцию `Parent()` два раза подряд,

```
Parent(10);
Parent(20);
```

то последний вызов породит ошибку: функция `Child()` уже определена. Это произошло потому, что `Child()` определяется внутри `Parent()`, и до ее определения управление программы фактически доходит дважды (при первом и втором вызовах `Parent()`). Поэтому-то интерпретатор и «протестует»: он не может второй раз добавить `Child()` в таблицу функций.

Условно определяемые функции. Предположим, у нас в программе где-то устанавливается переменная `$OS_TYPE` в значение `win`, если сценарий запущен под Windows 9x, и в `unix`, если под Unix. В отличие от Unix, в Windows нет такого понятия, как владелец файла, а значит, стандартная функция `chown()` (которая назначает владельца для указанного файла) там просто не имеет смысла. В некоторых версиях PHP для Windows ее может в этой связи вообще не быть. Однако, чтобы улучшить переносимость сценариев с одной платформы на другую (без изменения их кода!) можно написать следующую простую «обертку» для функции `chown()`:

Пример условно определяемой функции:


```
if($OS_TYPE=="win")
{ // Функция-заглушка
  function MyChOwn($fname,$attr)
  { // ничего не делает
    return 1;
  }
}
else
{ // Передаем вызов настоящей chown()
  function MyChOwn($fname,$attr)
  { return chown($fname,$attr);
  }
}
```

Если работаем под Windows, функция `MyChOwn()` ничего не делает и возвращает 1 как индикатор успеха, в то время как для Unix она просто вызывает оригинальную `chown()`.

7.4 Передача функций по ссылке и возврат функцией ссылки

Передача функций «по ссылке». Как таковая, передача функции по ссылке в PHP не поддерживается. Однако, т. к. это слишком часто может быть полезным, в PHP есть понятие «функциональной переменной». Пример:

```
function A($i) { echo "a $i\n"; }
function B($i) { echo "b $i\n"; }
function C($i) { echo "c $i\n"; }
$F="A"; // или $F="B" или $F="C"
$F(10); // вызов функции, имя которой хранится в $F
```

В PHP есть такая стандартная функция – `uasort()`, которая сортирует ассоциативный массив, заданный ее первым параметром, причем критерием сравнения для элементов этого массива служит функция, имя которой передано вторым параметром. Пример:

```
// Сравнение без учета регистра символов строк
function FCmp($a,$b)
{ return strcmp(tolower($a),tolower($b))
}
$a=array("b"=>"bbb", "a"=>"Aaa", "d"=>"ddd);
uasort($a,"FCmp"); // Сортировка без учета регистра символов
```

Здесь функция, имя которой получено со вторым параметром `uasort()`, должна иметь два аргумента, которые являются сравниваемыми значениями в массиве. В общем случае, функциональная переменная – это всего лишь переменная-строка, содержащая имя функции. Поскольку в PHP нет такого понятия, как области видимости для

функций (есть только области видимости для локальных переменных), то конфликтов это не порождает – одному имени может соответствовать не более одной функции. Такой подход не очень хорош, но он действительно работает.

Возврат функцией ссылки. До сих пор рассматривались функции, которые возвращают определенные значения – а именно, копии величин, использованных в инструкции `return` (это были именно копии, а не сами объекты). Пример:

```
$a=100;
function R()
{ global $a;   // объявляет $a глобальной
  return $a;   // возвращает значение, а не ссылку!
}
$b=R();
$b=0;          // присваивает $b, а не $a!
echo $a;       // выводит 100
```

Пусть необходимо, чтобы функция `R()` возвращала не величину, а *ссылку* на переменную `$a`, чтобы в дальнейшем с этой ссылкой можно было работать точно так же, как и с `$a`. Использование оператора `$b=&R()` не подходит, т. к. при этом мы получим в `$b` ссылку не на `$a`, а на ее копию. Если задействовать `return &$a`, то появится сообщение о синтаксической ошибке (PHP воспринимает `&` только в правой части оператора присваивания сразу после знака =).

Специальный синтаксис описания функции, возвращающей ссылку:

```
$a=100;
function &R()  // & — возвращает ссылку
{ global $a;   // объявляет $a глобальной
  return $a;   // возвращает значение, а не ссылку!
}
$b=&R();       // не забудьте & !!!
$b=0;          // присваивает переменной $a!
echo $a; // выводит 0. Это значит, что теперь $b — синоним $a
```

Нужно поставить `&` в двух местах: перед определением имени функции, а также в правой части оператора присваивания при вызове функции. Использовать амперсанд в инструкции `return` не нужно.

Тема 8 Строковые функции

8.1 Функции работы со строками
8.2 Работа с блоками текста

8.1 Функции работы со строками

Строки в PHP – одни из самых универсальных объектов. Любой объект можно упаковать в строку. Строка может содержать любые символы с кодами от 0 до 255 включительно. Нет специального маркера «конца строки», длина строки во внутреннем представлении PHP хранится отдельно. Для формирования и вставки непечатаемого символа в строку используется функция `chr()`. Из-за слабого контроля типов в PHP строка может содержать число, причем с ней можно работать, как с числом: прибавлять другие числа, умножать и т. д. При этом все преобразования (в десятичной системе) производятся автоматически. Существуют функции, преобразующие число, записанное в различных системах счисления в обычное представление, и наоборот.

Конкатенация строк. Конкатенация – это присоединение к одной строке другой. Оператор + следует применять только для сложения чисел. Для конкатенации строк используется специальный оператор «.» (точка). Оператор «.» всегда воспринимает свои операнды как строки и возвращает строку. В случае, если один из операндов не может быть переведен в строковое представление, т. е. если это массив или объект, то он воспринимается как строки `array` и `object` соответственно. Это правило применимо и не только при сцеплении строк, но и при передаче такого операнда в какую-нибудь стандартную функцию, которой требуется строка. Пример:

```
$a=array(10,20,30);
echo $a // Неожиданный результат «array»!
```

Есть и другой, более специализированный, способ конкатенации строк. Он обычно используется, когда значения строковых или числовых переменных перемежаются с обычными словами. Если, к примеру, в `$day` хранится текущее число, в `$month` – название месяца и в `$year` – год, то вывести строку вида «Сегодня 8 мая 2000 года» можно так:

```
echo "Сегодня $day $month $year года";
```

При этом в строку, вырабатываемую инструкцией `echo`, автоматически в нужных местах вставятся значения переменных.

Сравнение строк и инструкции if-else. Рассмотрим одно тонкое место в интерпретаторе PHP, касающееся немного неправильной работы со строками. Заключается оно в следующем: если используются операторы сравнения == и != (или любые другие, которые могут по-

требовать перевода строки в число) с операндами-строками, то результат, не всегда оказывается верным. Примеры:

```
$one=1       // число один
$zero=0      // присваиваем число ноль
if($one=="") echo 1 // очевидно, не равно – не выводит 1
if($zero=="") echo 3 // Вопреки ожиданиям печатает 3!
if(""==$zero) echo 4 // И это тоже не поможет!..
if("$zero"=="") echo 5
// Не работает в некоторых версиях PHP 3
if(strval($zero)=="") echo 6;
// Теперь правильно – не выводит 6
if($zero==="") echo 7
// Самый лучший способ, но не действует в PHP 3
```

В операциях сравнения пустая строка "" прежде всего трактуется как 0 (ноль) и уж затем – как «пусто»! Операнды сравниваются как строки только в том случае, если они оба – строки, в противном случае идет числовое сравнение. При этом пустая строка воспринимается как 0, как и любая другая, которую интерпретатору не удалось перевести в число.

При сравнении двух переменных-строк, нужно быть абсолютно уверенными, что их типы именно строковые, а не числовые. Это не распространяется на оператор === (оператор эквивалентности). Его использование заставляет интерпретатор *всегда* сравнивать величины и по значению, и по их типу. С точки зрения PHP `0==""`, но `0!===""`. Нужно всегда использовать === вместо `strval()`!

Существует одна стандартная ошибка, которая состоит в использовании функция `strpos($str,$what)`, которая возвращает позицию подстроки `$what` в строке `$str` или `false`, если подстрока не найдена. Пусть нужно проверить, встречается ли в строке `$str` подстрока `<?` (и напечатать «это PHP-программа», если встречается). Вариант

```
if(strpos($str,"<?")!=false)
echo "это PHP-программа";
```

не годится, если `<?` находится в самом начале строки (в этом случае не будет выдано сообщение, хотя подстрока в найдена, и функция возвратила 0, а не `false`). *Вместо проверки на неравенство, нужно проверять на неэквивалентность:*

```
if(strpos($str,"<?")!===false)
echo "это PHP-программа";
```

Оператор `!===` используется с константой `false`, а не с пустой строкой "". Для оператора `false!==""`, в то время как, `false==""`.

Функции для работы с одиночными символами:

- `string chr(int $code)`

  Возвращает строку из одного символа с кодом `$code`. Эта функция используется для вставки непечатаемых символов в строку – например, кода нуля или символа прогона страницы, а также при работе с бинарными файлами.
- `int ord(char $ch)`

  Функция возвращает код символа `$ch`.
- `int strrpos(string $where, char $what)`

  Функция ищет в строке `$where` последнюю позицию, в которой встречается символ `$what` (если `$what` – строка из нескольких символов, то выявляется только первый из них, остальные не играют никакой роли!). В случае, если искомый символ не найден, возвращается `false`

Функции отрезания пробелов:

- `string trim(string $st)`

  Функция возвращает копию `$st`, только с удаленными ведущими и концевыми пробельными символами. Под пробельными символами здесь и далее подразумеваются: пробел " ", символ перевода строки `\n`, символ возврата каретки `\r` и символ табуляции `\t`. Например, вызов `trim(" test\n ")` вернет строку `"test"`.
- `string ltrim(string $st)`

  Аналогична функции `trim()`, только удаляет исключительно ведущие пробелы, а концевые не трогает.
- `string chop(string $st)`

  Удаляет только концевые пробелы, ведущие не трогает.

Базовые функции:

- `int strlen(string $st)`

  Возвращает длину строки `$st`.
- `int strpos(string $where, string $what, int $fromwhere=0)`

  Пытается найти в строке `$where` подстроку (то есть последовательность символов) `$what` и в случае успеха возвращает позицию (индекс) этой подстроки в строке. Первый символ строки имеет индекс 0. Необязательный параметр `$fromwhere` можно задавать, если поиск нужно вести не с начала строки `$from`, а с какой-то другой позиции. В этом случае следует эту позицию передать в `$fromwhere`. Если подстроку найти не удалось, функция возвращает `false`. Проверяя результат вызова `strpos()` на `false` нужно использовать только оператор ===.

- `string substr(string $str, int $from [,int $length])`

Функция возвращает участок строки `$str`, начиная с позиции `$start` и длиной `$length`. Если `$length` не задана, то подразумевается подстрока от `$start` до конца строки `$str`. Если `$start` больше, чем длина строки, или же значение `$length` равно нулю, то возвращается пустая подстрока. Если передать в `$start` отрицательное число, то будет считаться, что это число является индексом подстроки, но только отсчитываемым от конца `$str` (например, –1 означает «начиная с последнего символа строки»). Параметр `$length`, если он задан, тоже может быть отрицательным. В этом случае последним символом возвращенной подстроки будет символ из `$str` с индексом `$length`, определяемым от конца строки.

- `int strcmp(string $str1, string $str2)`

Сравнивает две строки посимвольно (точнее, побайтово) и возвращает: 0, если строки полностью совпадают; –1, если строка `$str1` лексикографически меньше `$str2`; и 1, если, наоборот, `$str1` «больше» `$str2`. Так как сравнение идет побайтово, то регистр символов влияет на результаты сравнений.

- `int strcasecmp(string $str1, string $str2)`

Аналогична `strcmp()`, только при работе не учитывается регистр букв. С точки зрения этой функции `"ab"` и `"AB"` равны.

8.2 Работа с блоками текста

- `string str_replace(string $from, string $to, string $str)`

Заменяет в строке `$str` все вхождения подстроки `$from` (с учетом регистра) на `$to` и возвращает результат. Исходная строка, переданная третьим параметром, при этом не меняется. Пример замены всех символов перевода строки на их HTML- эквивалент – тэг
:

```
$st=str_replace("\n","<br>\n",$st)
```

То, что в строке `<br>\n` тоже есть символ перевода строки, никак не влияет на работу функции, т. е. функция производит лишь однократный проход по строке.

- `string nl2br(string $string)`

Заменяет в строке все символы новой строки `\n` на `<br>\n` и возвращает результат. Исходная строка не изменяется. Символы `\r`, которые присутствуют в конце строки текстовых файлов Windows, этой функцией никак не учитываются, а потому остаются на старом месте.

- `string WordWrap(string $st, int $width=75, string $break="\n")`

Функция разбивает блок текста `$st` на несколько строк, завершаемых символами `$break`, так, чтобы на одной строке было не более `$width` букв. Разбиение происходит по границе слова, так что текст остается читаемым. Возвращается получившаяся строка с символами перевода строки, заданными в `$break`. Пример форматирования текста по ширине поля 60 символов, предварив каждую строку префиксом `">"`:

```
function Cite($OurText, $prefix="> ")
{ $st=WordWrap($OurText, 60-strlen($prefix), "\n");
  $st=$prefix.str_replace("\n","\n$prefix",$st);
  return $st;
}
```

- `string strip_tags (string $str [, string $allowable_tags])`

Функция удаляет из строки все тэги и возвращает результат. В параметре `$allowable_tags` можно передать тэги, которые не следует удалять из строки. Они должны перечисляться вплотную друг к другу. Пример:

```
$st="
<b>Жирный текст</b>
<tt>Моноширинный текст</tt>
<a href=http://www.dklab.ru>Ссылка</a>";
echo "Исходный текст: $st";
echo "<hr>После удаления тэгов:
".strip_tags($st,"<a><b>")."<hr>";
```

Тэги <a> и <b> не были удалены (ровно как и их парные закрывающие), в то время как <tt> исчез.

- `string str_repeat(string $st, string $number)`

Функция «повторяет» строку `$st` `$number` раз и возвращает объединенный результат. Пример:

```
echo str_repeat("test!",3); // выводит test!test!test!
```

Функции для преобразований символов:

- `string strtr(string $str, string $from, string $to)`

Функция в строке `$str` заменяет все символы, встречающиеся в `$from`, на их «парные» (то есть расположенные в тех же позициях, что и во `$from`) из `$to`.

- `string UrlEncode(string $st)`

Функция URL-кодирует строку $st и возвращает результат. Эту функцию удобно применять, если нужно динамически сформировать ссылку <a href=...> на какой-то сценарий, но не уверены, что его параметры содержат только алфавитно-цифровые символы. Пример:

```
echo "<a
href=/script.php?param=".UrlEncode($UserData);
```

Если переменная $UserData включает символы =, & или даже пробелы, все равно сценарию будут переданы корректные данные.

- `string UrlDecode(string $st)`

Производит URL-декодирование строки. Используется редко, т.к. PHP и так умеет перекодировать входные данные автоматически.

- `string RawUrlEncode(string $st)`

Почти полностью аналогична `UrlEncode()`, но только пробелы не преобразуются в +, как это делается при передаче данных из формы, а воспринимаются как обычные неалфавитно-цифровые символы.

- `string RawUrlDecode(string $st)`

Аналогична `UrlDecode()`, но не воспринимает + как пробел.

- `string HtmlSpecialChars(string $str)`

Заменяет в строке символы: &, <, >, кавычки на их HTML-эквиваленты, так, чтобы они выглядели на странице «самими собой». Пример применения этой функции – формирование параметра value в элементах формы, чтобы не было проблем с кавычками, или вывод сообщения в гостевой книге, если вставлять тэги пользователю запрещено. Пусть содержимое книги хранится в массиве $Book. Пример печати содержимого гостевой книги, чтобы тэги не воспринимались браузером как описания форматирования:

```
<?foreach($Book as $k=>$v) {?>
Имя: <?=$v['name']?><br>
Текст: <?=HtmlSpecialChars($v['text'])?> <hr>
<?}?>
```

- `string StripSlashes(string $st)`

Заменяет в строке $st некоторые предваренные слэшем символы на их однокодовые эквиваленты. Это относится к символам: ", ', \.

- `string AddSlashes(string $st)`

Вставляет слэши только перед символами: ', " и \.

Функции изменения регистра:

- `string strtolower(string $str)`

Преобразует строку в нижний регистр.

- `string strtoupper(string $str)`

Переводит строку в верхний регистр.

Функции форматных преобразований. Переменные в строках PHP интерполируются, поэтому практически всегда задача «смешивания» текста со значениями переменных не является проблемой:

```
echo "Привет, $name! Вам $age лет.";
```

Язык PHP поддерживает ряд функций, использующих такой же синтаксис, как и их Си-эквиваленты.

- `string sprintf(string $format [, mixed args, ...])`

Функция возвращает строку, составленную на основе строки форматирования, содержащей некоторые специальные символы, которые будут впоследствии заменены на значения соответствующих переменных из списка аргументов. Строка форматирования $format может включать в себя команды форматирования, предваренные символом %. Все остальные символы копируются в выходную строку как есть. Каждый спецификатор формата (то есть, символ % и следующие за ним команды) соответствует одному параметру, указанному после параметра $format. Если нужно поместить в текст % как обычный символ, необходимо его удвоить:

```
echo sprintf("Значение %d%%",$d);
```

Каждый спецификатор формата включает максимум пять элементов (в порядке их следования после символа %):

- Необязательный спецификатор размера поля, который указывает, сколько символов будет отведено под выводимую величину. В качестве символов-заполнителей (если значение имеет меньший размер, чем размер поля для его вывода) может использоваться пробел или 0, по умолчанию подставляется пробел. Можно задать любой другой символ-наполнитель, если указать его в строке форматирования, предварив апострофом '.
- Опциональный спецификатор выравнивания, определяющий, будет результат выровнен по правому или по левому краю поля. По умолчанию производится выравнивание по правому краю, однако можно указать и левое выравнивание, задав символ – (минус).
- Необязательное число, определяющее размер поля для вывода величины. Если результат не будет в поле помещаться, то он «вылезет» за края этого поля, но не будет усечен.
- Необязательное число, предваренное точкой ".", предписывающее, сколько знаков после запятой будет в результирующей строке. Этот спецификатор учитывается только в том случае, если происходит вывод числа с плавающей точкой, в противном случае он игнорируется.
- Обязательный спецификатор типа величины, которая будет помещена в выходную строку: b – очередной аргумент из списка выво-

дится как двоичное целое число; c – выводится символ с указанным в аргументе кодом; d – целое число; f – число с плавающей точкой; o – восьмеричное целое число; s – строка символов; x – шестнадцатеричное целое число с маленькими буквами a–z; X – шестнадцатеричное число с большими буквами A–Z.

Примеры:

```
$money1 = 68.75;
$money2 = 54.35;
$money = $money1 + $money2;
// echo $money выведет "123.1"...
$formatted = sprintf ("%01.2f", $money);
// echo $formatted выведет "123.10"!
// вывода целого числа, предваренного нужным количеством нулей:
$isodate=sprintf("%04d-%02d-%02d",  $year,  $month,
$day);
```

- `void printf(string $format [, mixed args, ...])`

Делает то же, что и sprintf(), только результирующая строка не возвращается, а направляется в браузер пользователя.

- `string number_format(float $number,`
  `int $decimals, string $dec_point=".",`
  `string $thousands_sep=",");`

Функция форматирует число с плавающей точкой с разделением его на триады с указанной точностью. Она может быть вызвана с двумя или четырьмя аргументами, но не с тремя! Параметр $decimals задает, сколько цифр после запятой должно быть у числа в выходной строке. Параметр $dec_point представляет собой разделитель целой и дробной частей, а параметр $thousands_sep – разделитель триад в числе (если указать на его месте пустую строку, то триады не отделяются друг от друга).

В PHP существует еще несколько функций для выполнения форматных преобразований, среди них – sscanf() и fscanf(), которые часто применяются в Си.

Работа с бинарными данными. Строки могут содержать любые, в том числе и бинарные, данные (то есть, символы с кодами, меньшими 32). Для работы с такими строками иногда удобно использовать функции pack() и unpack().

- string pack(string $format [,mixed $args, ...])

Функция pack() упаковывает заданные аргументы в бинарную строку. Формат параметров, а также их количество, задается при помощи строки $format, которая представляет собой набор однобуквенных спецификаторов форматирования – наподобие тех, которые указываются в sprintf(), но только без знака %. После каждого спецификатора может стоять число, которое отмечает, сколько информации будет обработано данным спецификатором. А именно, для форматов a, A, h и H число задает, какое количество символов будет помещено в бинарную строку из тех, что находятся в очередном параметре-строке при вызове функции (то есть, определяется размер поля для вывода строки). В случае @ оно определяет абсолютную позицию, в которую будут помещены следующие данные. Для всех остальных спецификаторов следующие за ними числа задают количество аргументов, на которые распространяется действие данного формата. Вместо числа можно указать *, в этом случае подразумевается, что спецификатор действует на все оставшиеся данные. Список спецификаторов формата: a – строка, свободные места в поле заполняются символом с кодом 0; A – строка, свободные места заполняются пробелами; h – шестнадцатеричная строка, младшие разряды в начале; H – шестнадцатеричная строка, старшие разряды в начале; c – знаковый байт (символ); C – беззнаковый байт; s – знаковое короткое целое (16 бит); S – беззнаковое короткое целое; n – беззнаковое целое (16 битов, старшие разряды в конце); v – беззнаковое целое (16 битов, младшие разряды в конце); i – знаковое целое; I – беззнаковое целое; l – знаковое длинное целое; L – беззнаковое длинное целое; N – беззнаковое длинное целое; V – беззнаковое целое (32 бита, младшие разряды в конце); f – число с плавающей точкой; d – число с плавающей точкой двойной точности; x – символ с нулевым кодом; X – возврат назад на 1 байт; @ –заполнение нулевым кодом до заданной абсолютной позиции.

Пример:

```
// Целое, целое, все остальное – символы
$bindata = pack("nvc*", 0x1234, 0x5678, 65, 66);
```

В строке `$bindata` будет содержаться 6 байтов в такой последовательности: 0x12, 0x34, 0x78, 0x56, 0x41, 0x42 (в шестнадцатеричной системе счисления).

- `array unpack(string $format, string $data)`

Функция распаковывает строку `$data`, пользуясь информацией о формате `$format`. Возвращает она ассоциативный массив, содержащий элементы распакованных данных. В строке `$format` после каждого спецификатора (или после завершающего его числа) должно «впритык» следовать имя ключа в ассоциативном массиве. Разделяются параметры при помощи символа /. Пример:

```
$array=unpack("c2chars/nint", $bindata);
```

В результирующий массив будут записаны элементы с ключами: `chars1`, `chars2` и `int`. Если после спецификатора задано число, то к

имени ключа будут добавлены номера 1, 2 и т. д., т. е. в массиве появятся несколько ключей, отличающихся суффиксами.

Хэш-функции

- `string md5(string $st)`

Возвращает хэш-код строки `$st`, основанный на алгоритме корпорации RSA Data Security под названием «MD5 Message-Digest Algorithm». Хэш-код – это строка, практически уникальная для каждой из строк `$st`. То есть вероятность того, что две *разные* строки, переданные в `$st`, дадут нам *одинаковый* хэш-код, стремится к нулю. Если длина строки `$st` может достигать нескольких тысяч символов, то ее MD5-код занимает максимум 32 символа.

Хэш-код и алгоритм MD5 используется для проверки паролей на истинность. Пусть, например, есть система со многими пользователями, каждый из которых имеет свой пароль. Можно хранить все пароли в обычном виде, или зашифровать их каким-нибудь способом, но тогда велика вероятность того, что этот файл с паролями украдут. Если пароли были зашифрованы, то, зная метод шифрования, не составит особого труда их раскодировать. Однако можно поступить другим способом, при использовании которого даже если файл с паролями украдут, расшифровать его будет математически невозможно. Сделаем так: в файле паролей будем хранить не сами пароли, а их (MD5) хэш-коды. При попытке какого-либо пользователя войти в систему мы вычислим хэш-код только что введенного им пароля и сравним его с тем, который записан у нас в базе данных. Если коды совпадут, значит, все в порядке, а если нет – что ж, извините... При вычислении хэш-кода какая-то часть информации о строке `$st` безвозвратно теряется. И именно это позволяет не опасаться, что файл паролей можно расшифровать. Ведь в нем нет самих паролей, нет даже их каких-то связных частей!

Алгоритм MD5 специально был изобретен для того, чтобы обеспечить описанную выше схему. Так как все же есть вероятность того, что у разных строк MD5-коды совпадут, то, чтобы не дать возможность войти в систему, перебирая пароли с бешеной скоростью, алгоритм MD5 работает довольно медленно. И его нельзя никак убыстрить, потому что это будет уже не MD5. Так что даже на самых мощных компьютерах вряд ли получится перебирать более нескольких тысяч паролей в секунду, а это совсем маленькая скорость.

- `int crc32(string $str)`

Функция вычисляет 32-битную контрольную сумму строки `$str`. То есть, результат ее работы – 32-битное (4-байтовое) целое число. Эта функция работает гораздо быстрее `md5()`, но в то же время выдает гораздо менее надежные «хэш-коды» для строки. Алгоритм `crc32` имеет неизмеримо большую предсказуемость, чем MD5.

- `string crypt(string $str [,string $salt])`

Алгоритм шифрования DES до недавнего времени был стандартным для всех версий Unix и использовался для кодирования паролей пользователей (тем же самым способом, о котором мы говорили при рассмотрении функции md5()). Но в последнее время MD5 постепенно начал его вытеснять. Хэш-код для одной и той же строки, но с различными значениями `$salt` (это должна быть обязательно двухсимвольная строка) дает разные результаты. Если параметр `$salt` пропущен, PHP сгенерирует его случайным образом:

```
$st="This is the test";
echo crypt($st)."<br>";
// можем получить, например, 7N8JKLKbBWEhg
echo crypt($st)."<br>";
// а здесь появится, например, Jsk746pawBOA2
```

Два одинаковых вызова `crypt()` без второго параметра выдают совершенно разные хэш-коды.

Сброс буфера вывода

- `void flush()`

Функция отправляет содержимое буфера echo в браузер пользователя. Обычно при использовании echo данные не прямо сразу отправляются клиенту, а накапливаются в специальном буфере, чтобы потом транспортироваться большой «пачкой». Так получается быстрее. Однако, иногда бывает нужно досрочно отправить все данные из буфера пользователю, например, если что-то выводится в реальном времени.

Тема 9 Математические функции

9.1 Встроенные константы
9.2 Встроенные функции

9.1 Встроенные константы

В PHP представлен полный набор математических функций. Встроенные констнты приведены в таблице 9.1.

Таблица 9.1 – Встроенные константы

Константа	Значение	Пояснение
M_PI	3,14159265358979323846	Число π
M_E	2,7182818284590452354	*e*
M_LOG2E	1,4426950408889634074	*log*2(e)
M_LOG10E	0,43429448190325182765	*lg*(e)
M_LN2	0,69314718055994530942	*ln*(2)
M_LN10	2,30258509299404568402	*ln* (10)
M_PI_2	1,57079632679489661923	π□/2
M_PI_4	0,78539816339744830962	π□/4
M_1_PI	0,31830988618379067154	1/π
M_2_PI	0,63661977236758134308	2/π
M_SQRTPI	1,77245385090551602729	$\sqrt{\pi}$
M_2_SQRTPI	1,12837916709551257390	$2/\sqrt{\pi}$
M_SQRT2	1,41421356237309504880	$\sqrt{2}$
M_SQRT3	1,73205080756887729352	$\sqrt{3}$
M_SQRT1_2	0,70710678118654752440	$1/\sqrt{2}$
M_LNPI	1,14472988584940017414	*ln*(π)
M_EULER	0,57721566490153286061	Постоянная Эйлера

9.2 Встроенные функции

Функции округления:

- `mixed abs(mixed $number)`

Возвращает модуль числа. Тип параметра `$number` может быть `float` или `int`, а тип возвращаемого значения всегда совпадает с типом этого параметра.

- `double round(double $val)`

Округляет `$val` до ближайшего целого, например:

```
$foo = round(3.4); // $foo == 3.0
$foo = round(3.5); // $foo == 4.0
```


- `int ceil(float $number)`

Возвращает наименьшее целое число, не меньшее `$number`.

- `int floor(float $number)`

Возвращает максимальное целое число, не превосходящее `$number`.

Случайные числа:

- `int mt_rand(int $min=0, int $max=RAND_MAX)`

Функция возвращает случайное число от 0 до `RAND_MAX` (эта константа задает максимально допустимое случайное число). Перед первым вызовом этой функции занужно пустить `mt_srand()`.

- `void mt_srand(int $seed)`

Настраивает генератор случайных чисел на новую последовательность. Числа, генерируемые `mt_rand()`, достаточно равновероятны, но у них есть недостаток: последовательность сгенерированных чисел будет одинакова если сценарий вызвать несколько раз подряд. Функция `mt_srand()` выбирает новую последовательность на основе параметра `$seed`, практически непредсказуемым образом.

- `int mt_getrandmax()`

Возвращает максимальное число, которое может быть сгенерировано функцией `mt_rand()`, т.е. константу `RAND_MAX`.

Перевод в различные системы счисления

- `string base_convert(string $number, int $frombase, int $tobase)`

Переводит число `$number` (заданное как строка в системе счисления по основанию `$frombase`) в систему по основанию `$tobase`. Параметры `$frombase` и `$tobase` могут принимать значения только от 2 до 36 включительно. Пример:`echo base_convert("FF",16,2);`

- `int bindec(string $binary_string)`

Преобразует двоичное число, заданное в строке `$binary_string`, в десятичное число.

- `string decbin(int $number)`

Возвращает строку, представляющую собой двоичное представление целого числа `$number`. Максимальное число, которое может быть преобразовано, равно 2 147 483 647, (31 единичка в двоичной системе). Для восьмеричной и шестнадцатеричной систем существует такие же функции, называются они так же, только вместо `"bin"` подставляется соответственно `"oct"` и `"hex"`.

Минимум и максимум:

- `mixed min(mixed $arg1 [int $arg2, ..., int $argn])`

Функция возвращает наименьшее из чисел, заданных в ее аргументах. Различают два способа вызова этой функции: с одним параметром или с несколькими. Если указан лишь один параметр (первый), то он обязательно должен быть массивом и возвращается минимальный элемент этого массива. В противном случае первый (и остальные) аргументы трактуются как числа с плавающей точкой, они сравниваются, и возвращается наименьшее. Тип возвращаемого значения выбирается так: если хотя бы одно из чисел, переданных на вход, задано в формате с плавающей точкой, то и результат будет с плавающей точкой, в противном случае результат будет целым числом. *С помощью этой функции нельзя лексикографически сравнивать строки – только числа.*

- `mixed max(mixed $arg1 [int $arg2, ..., int $argn])`

  Функция ищет максимальное значение (аналогична `min()`).

Степенные функции:

- `float sqrt(float $arg)`

  Возвращает квадратный корень из аргумента. Если аргумент отрицателен, генерируется предупреждение.

- `float log(float $arg)`

  Возвращает натуральный логарифм аргумента. В случае недопустимого числа печатает предупреждение, но не завершает программу.

- `float exp(float $arg)`

  Возвращает e (2,718281828…) в степени `$arg`.

- `float pow(float $base, float $exp)`

  Возвращает `$base` в степени `$exp`.

Тригонометрия:

- `float acos(float $arg)` // арккосинус
- `float asin(float $arg)` // арксинус
- `float atan(float $arg)` // арктангенс
- `float atan2(float $y, float $x)`

  Возвращает арктангенс величины `$y/$x`, но с учетом той четверти, в которой лежит точка (`$x`, `$y`). Эта функция возвращает результат в радианах, принадлежащий отрезку от – ∞ до +∞. Пример:

```
$alpha=atan2(1,1); // $alpha==pi/4
$alpha=atan2(-1,-1); // $alpha==-3*pi/4
```

- `float sin(float arg)` // синус
- `float cos(float $arg)` // косинус
- `float tan(float arg)` // тангенс

  Для синуса, косинуса, тангенса аргумент задается в радианах.

- `double pi()` // число π.

Тема 10 Работа с файлами и каталогами, запуск внешних программ

10.1 Функции работы с файлами
10.2 Блокирование файла
10.3 Функции работы с каталогами
10.4 Запуск внешних программ

10.1 Функции работы с файлами

При работе с файлами интерпретатору PHP все равно, какие слэши использовать: прямые или обратные. Он в любом случае переведет их в ту форму, которая требуется ОС, а функции по работе с полными именами файлов также не будут против «чужих» слэшей.

Можно работать с файлами на удаленных серверах Web в точности так же, как и со своими собственными. Если имени файлу предшествует строка `http://` или `ftp://`, то PHP понимает, что нужно установить сетевое соединение и работать именно с ним, а не с файлом. При этом в программе такой файл ничем не отличается от обычного.

Текстовые и бинарные файлы. Во многих языках программирования для работы с текстовыми файлами применяется некоторый трюк. В Unix-системах для отделения одной строки файла от другой используется один специальный символ – его принято обозначать \n.

Здесь \n обозначает именно *один* символ, т. е., один байт. Когда PHP встречает комбинацию \n в строке (например, «это \n тест»), он воспринимает ее как один байт. В то же время, в строке, заключенной в апострофы, комбинация \n не имеет никакого специального назначения и обозначает буквально «символ \, за которым идет символ n».

В Windows для разделения строк применяется не один, а два символа, следующих подряд, – `\r\n`. Для того чтобы языки программирования были лучше переносимы с одной операционной системы на другую, при чтении текстовых файлов комбинация `\r\n` преобразуется «на лету» в один символ `\n`, так что программа и не замечает, что формат файла не такой, как в Unix. Поэтому, если например, прочитать содержимое всего текстового файла в строку, то длина такой строки окажется меньше физического размера файла – ведь из нее «съелись» некоторые символы `\r`. Это относится к системе Windows. При записи строки в текстовый файл происходит в точности наоборот: один `\n` становится на диске парой `\r\n`.

Практически во всех языках программирования можно отключить режим автоматической трансляции `\r\n` в один `\n`. Обычно для этого используется вызов специальной функции, который говорит, что для

указанного файла нужно применять *бинарный режим* ввода/вывода, когда все байты читаются, как есть.

Так как PHP был написан целиком на Си, а Си использует трансляцию символов перевода строк, то описанная техника работает и в PHP. Однако есть один очень опасный момент. Интерпретатор может работать с файлами в режиме трансляции символа перевода строки. Если файл открыт в режиме бинарного чтения/записи, то PHP все равно, что читается или пишется. Можно считать содержимое бинарного файла (например, GIF-рисунка) в обычную строковую переменную, а потом записать эту строку в другой файл, и при этом информация нисколько не исказится. При чтении текстового файла в Windows получите символы `\r\n` в конце строки вместо одного `\n`.

Открытие файла. Работа с файлами в PHP разделяется на три этапа. Сначала файл открывается в нужном режиме, при этом возвращается целое число, служащее идентификатором открытого файла (дескриптор файла). Затем используются команды работы с файлом (чтение или запись, или и то и другое), причем они «привязаны» уже к дескриптору файла, а не к его имени. После этого файл лучше всего закрыть (хотя это можно и не делать, поскольку PHP автоматически закрывает все файлы по завершении сценария).

- `int fopen (string $filename, string $mode, bool $use_include_path=false)`

Функциця открывает файл с именем `$filename` в режиме `$mode` и возвращает дескриптор открытого файла. В случае ошибки функция возвращает `false`. Дескриптор 0 в системе соответствует стандартному потоку ввода, он никогда не будет открыт функцией `fopen()` (во всяком случае, пока не будет закрыт нулевой дескриптор). Необязательный параметр `$use_include_path` говорит PHP о том, что, если задано относительное имя файла, его следует искать также и в списке путей, используемом инструкциями `include` и `require`. Обычно этот параметр не используют. Параметр `$mode` может принимать следующие значения:

`r` – файл открывается только для чтения. Если файла не существует, вызов регистрирует ошибку. После удачного открытия указатель файла устанавливается на его первый байт, т. е. на начало;

`r+` – файл открывается одновременно на чтение и запись. Указатель текущей позиции устанавливается на его первый байт. Если файла *не существует*, возвращается `false`. Если в момент записи указатель файла установлен где-то в середине файла, то данные запишутся прямо поверх уже имеющихся, а не «раздвинут» их, при необходимости увеличив размер файла.

`w` – создает новый пустой файл. Если на момент вызова уже был файл с таким именем, то он предварительно уничтожается.

`w+` – аналогичен `r+`, но если файла не существовало, создает его. После этого с файлом можно работать как в режиме чтения, так и записи. Если файл существовал до момента вызова, его содержимое удаляется;

`a` – открывает существующий файл в режиме записи, и при этом сдвигает указатель текущей позиции за последний байт файла. Этот режим используется для дополнения файла.

`a+` – открывает файл в режиме чтения и записи, указатель файла устанавливается на конец файла, при этом содержимое файла не уничтожается. Отличается от `a` тем, что если файла изначально не существовало, то он создается. Этот режим полезен, если нужно что-то дописать в файл, но неизвестно, создан ли уже такой файл;

В конце любой из строк `r`, `w`, `a`, `r+`, `w+` и `a+` может находиться еще один необязательный символ – `b` или `t`. Если указан `b` (или не указан вообще никакой), то файл открывается в режиме бинарного чтения/записи. Если же это `t`, то для файла устанавливается режим трансляции символа перевода строки, т. е. он воспринимается как текстовый.

Пример работы с текстовыми файлами:

```
<?
// Получает в параметрах строку и возвращает через пробел коды
// символов, из которых она состоит
function MakeHex($st)
{ for($i=0; $i<strlen($st); $i++)
     $Hex[]=sprintf("%2X",ord($st[$i]));
return join(" ",$Hex);
}
// Открываем файл разными способами
$f=fopen("test.php","r"); // бинарный режим
echo MakeHex(fgets($f,100)),"<br>\n";
$f=fopen("test.php","rt"); // текстовый режим
echo MakeHex(fgets($f,100)),"<br>\n";
?>
```

Первая строчка файла `test.php` состоит всего из двух символов – это `<` и `?`. За ними должен следовать маркер конца строки. Сценарий показывает, как выглядит этот маркер, т. е. состоит ли он из одного или двух символов. Результат работы сценария:

```
3C 3F 0D 0A
3C 3F 0A
```

Бинарное и текстовое чтение дали разные результаты! В последнем случае произошла трансляция маркера конца строки.

Можно предварять имя файла строкой `http://` или `ftp://`, при этом будет осуществляться доступ к файлу с удаленного хоста. В случае HTTP-доступа PHP открывает соединение с указанным сервером, а также посылает ему нужные заголовки: `Post` и `GET`. После чего при помощи файлового дескриптора из удаленного файла можно читать обычным образом, например с помощью функции `fgets()`.

Если открывается FTP-файл, то в него можно производить либо запись, либо читать из него, но не и то и другое одновременно. Кроме того, лучше не использовать обратные слэши \ в именах файлов. При использовании обратного слэша нужно его удвоить, потому что в строках он воспринимается как спецсимвол:

```
$fp = fopen ("c:\\windows\\hosts", "r");
```

Этот способ не переносим между операционными системами. Лучше не использовать его! Примеры:

```
// Открывает файл на чтение
$f=fopen("/home/file.txt","r")or die("Ошибка!");
// Открывает HTTP-соединение на чтение
$f=fopen("http://www.php/","r") or die("Ошибка!");
// Открывает FTP-соединение с указанием логина и пароля для записи
$f = fopen("ftp://user:password@example.com/", "w")
or die("Ошибка!");
```

Конструкция or die(). Эта конструкция применяется при работе с файлами. Оператор `or` имеет очень низкий приоритет (даже ниже, чем у =), поэтому всегда выполняется уже после присваивания.

Если файл открыть не удалось, `fopen()` возвращает `false`, а значит, осуществляется вызов функции `die()`. Нельзя просто заменить `or` на, казалось бы равнозначный ему оператор `||`, потому что последний имеет гораздо более высокий приоритет – выше, чем у =. Таким образом, в результате вызова функции

```
$f=fopen("/home/file.txt", "r") || die("Ошибка!");
```

в действительности будет выполнено

```
$f=(fopen("/home/file.txt","r") || die("Ошибка!"));
```

Это не совсем то, что нам нужно.

Безымянные временные файлы. Иногда приходится работать с временными файлами, которые при завершении программы хотелось бы удалить. При нужно знать лишь файловый дескриптор, а не имя временного файла. Для создания таких объектов в PHP предусмотрена специальная функция.

- `int tmpfile()`

Создает новый файл с уникальным именем и открывает его на чтение и запись. В дальнейшем вся работа должна вестись с возвращенным файловым дескриптором, потому что имя файла недоступно. Пространство, занимаемое временным файлом, автоматически освобождается при его закрытии и при завершении работы программы.

Закрытие файла. После работы файл лучше всего закрыть (несмотря на то, что PHP сам закрывает все файлы по завершении работы сценария). Для этого используется следующая функция:

- `int fclose(int $fp)`

Функция закрывает файл и возвращает `false`, если файл закрыть не удалось. В противном случае возвращает значение «истина».

Необходимо *всегда* закрывать FTP- и HTTP-соединения, потому что в противном случае «беспризорный» файл приведет к неоправданному простою канала и излишней загрузке сервера. Кроме того, успешно закрыв соединение, можно быть уверены в том, что все данные были доставлены без ошибок.

Особенно своевременное закрытие критично при использовании FTP-файла в режиме записи, когда вывод программы для ускорения буферизуется. Не закрыв файл, вообще нельзя быть увереным, что буфер вывода очистился, а значит, файл записался на удаленную машину верно.

Чтение и запись. Для каждого открытого файла (точнее, для каждого файлового дескриптора, ведь один и тот же файл может быть открыт несколько раз, т. е. с ним может быть связано сразу несколько дескрипторов) система хранит определенную величину, которая называется текущей позицией ввода-вывода, или указатель файла. Функции чтения и записи файлов работают именно с этой позицией. А именно, функции чтения читают блок данных, начиная с этой позиции, а функции записи – записывают, также отсчитывая от нее. Если указатель файла установлен за последним байтом и осуществляется запись, то файл автоматически увеличивается в размере. После того как файл успешно открыт, из него (при помощи дескриптора файла) можно читать, а также, при соответствующем режиме открытия, писать. Обмен данными осуществляется через строки, начиная с позиции указателя файла.

Блочные чтение/запись.

- `string fread(int $f, int $numbytes)`

Читает из файла `$f` `$numbytes` символов и возвращает строку этих символов. После чтения указатель файла продвигается к следующим после прочитанного блока позициям. Если `$numbytes` больше, чем можно прочитать из файла (например, раньше достигается конец файла), возвращается то, что удалось считать. Этот прием можно использовать, если нужно считать в строку файл целиком. Для этого нужно задайть в `$numbytes` очень большое число (например, сто тысяч). Но нужно заботиться об экономии памяти в системе, так поступать не рекомендуется, потому что в некоторых версиях PHP передача большой

длины строки во втором параметре `fread()` вызывает первоначальное выделение этой памяти в соответствии с запросом (даже если строка гораздо короче). Конечно, потом лишняя память освобождается, но все же ее может и не хватить для начального выделения.

- `int fwrite(int $f, string $st)`

Записывает в файл `$f` все содержимое строки `$st`. При работе с текстовыми файлами все `\n` автоматически преобразуются в тот разделитель строк, который принят в операционной системе.

Построчные чтение/запись.

- `string fgets(int $f, int $length)`

Читает из файла одну строку, заканчивающуюся символом новой строки `\n`. Этот символ также считывается и включается в результат. Если строка в файле занимает больше `$length-1` байтов, то возвращаются только ее `$length-1` символов.

- `int fputs(int $f, string $st)`

Эта функция – полный аналог `fwrite()`.

Чтение CSV-файла.

- `list fgetcsv(int $f, int $length, char $delim=',')`

Функция читает одну строку из файла Excel, записанного в формате csv, заданного дескриптором `$f`, и разбивает ее по символу `$delim`. Параметр `$delim` должен быть строкой из одного символа, в противном случае принимается во внимание только первый символ этой строки. Функция возвращает получившийся список или `false`, если строки кончились. Параметр `$length` задает максимальную длину строки точно так же. Пустые строки в файле *не* игнорируются, а возвращаются как список из одного элемента – пустой строки.

Пример использования функции:

```
$f=fopen("file.csv","r") or die("Ошибка!");
for($i=0; $data=fgetcsv($f, 1000, ";"); $i++) {
$num = count($data);
if($num==1 && $data[0]==="") continue;
echo "<h3>Строка номер $i ($num полей):</h3>";
for($c=0; $c<$num; $c++)
print "[$c]: $data[$c]<br>";
}
fclose($f);
```

Положение указателя текущей позиции.

- `int feof(int $f)`

Функция возвращает `true`, если достигнут конец файла. Эта функция чаще всего используется в следующем контексте:


```
$f=fopen(: "myfile.txt","r");
while(!feof($f))
{ $st=fgets($f);
// обрабатываем очередную строку $st
//...
}
fclose($f);
```

Лучше избегать подобных конструкций, т. к. в случае больших файлов они довольно медлительны. Лучше читать файл целиком при помощи `File()` или `fread()`, если нужен доступ к каждой строке этого файла, а не только к нескольким первым!

- `int fseek(int $f, in $offset, int $whence=SEEK_SET)`

Устанавливает указатель файла на байт со смещением `$offset` (от начала файла, от его конца или от текущей позиции, в зависимости от параметра `$whence`). Эта функция может не сработать, если дескриптор `$f` ассоциирован не с обычным локальным файлом, а с соединением HTTP или FTP. Параметр `$whence` задает, с какого места отсчитывается смещение `$offset`. В PHP для этого существуют три константы, равные, соответственно, 0, 1 и 2: `SEEK_SET` – устанавливает позицию начиная с начала файла; `SEEK_CUR` – отсчитывает позицию относительно текущей позиции; `SEEK_END` – отсчитывает позицию относительно конца файла. В случае использования последних двух констант параметр `$offset` может быть отрицательным (а при применении `SEEK_END` он будет отрицательным *наверняка*). В случае успешного завершения функция возвращает 0, а в случае неудачи – 1.

- `int ftell(int $f)`

Возвращает положение указателя файла.

Функции для определения типов файлов. PHP имеет набор вспомогательных функций для работы с файлами. Они отличаются тем, что работают не с файловыми идентификаторами, а непосредственно с их именами.

Определение типа файла

- `bool file_exists(string $filename)`

Возвращает `true`, если файл с именем `$filename` существует на момент вызова.

- `string filetype(string $filename)`

Возвращает строку, которая описывает тип файла с именем `$filename`. Если такого файла не существует, возвращает `false`. После вызова строка будет содержать одно из следующих значений: `file` – обычный файл; `dir` – каталог; `link` – символическая ссылка; `fifo` –

fifo-канал; block – блочно-ориентированное устройство; char – символьно-ориентированное устройство; unknown – неизвестный тип файла.

- bool is_file(string $filename)
  Возвращает true, если $filename – обычный файл.
- bool is_dir(string $filename)
  Возвращает true, если $filename – каталог.
- bool is_link(string $filename)
  Возвращает true, если $filename – символическая ссылка.

Определение возможности доступа.

- bool is_readable(string $filename)
  Возвращает true, если файл может быть открыт для чтения.
- bool is_writeable(string $filename)
  Возвращает true, если в файл можно писать.
- bool is_executable(string $filename)
  Возвращает true, если файл – исполняемый.

Определение параметров файла.

- array stat(string $filename)

Функция собирает вместе всю информацию, выдаваемую операционной системой для указанного файла, и возвращает ее в виде массива. Этот массив всегда содержит следующие элементы с указанными ключами: 0 – устройство; 1 – номер узла inode; 2 – атрибуты защиты файла; 3 – число синонимов («жестких» ссылок) файла; 4 – идентификатор uid владельца; 5 – идентификатор gid группы; 6 – тип устройства; 7 – размер файла в байтах; 8 – время последнего доступа в секундах, прошедших с 1 января 1970 года; 9 – время последней модификации содержимого файла; 10 – время последнего изменения атрибутов файла; 11 – размер блока; 12 – число занятых блоков.

Специализированные функции.

Для того чтобы каждый раз не возиться с вызовом stat() и разбором выданного массива, разработчики PHP предусмотрели несколько простых функций, которые сразу возвращают какой-то один параметр файла. Кроме того, если объекта (обычно файла), для которого вызвана какая-либо из ниже перечисленных функций, не существует, эта функция возвратит false.

- int fileatime(string $filename)

Возвращает время последнего доступа к файлу. Время выражается в количестве секунд, прошедших с 1 января 1970 года. Если файл не обнаружен, возвращает false.

- int filemtime(string $filename)

Возвращает время последнего изменения файла или false в случае отсутствия файла.

- int filectime(string $filename)
  Возвращает время создания файла.
- int filesize(string $filename)

Возвращает размер файла в байтах или false, если файла не существует.

- int touch(string $filename [, int $timestamp])

Устанавливает время модификации указанного файла $filename равным $timestamp (в секундах, прошедших с 1 января 1970 года). Если второй параметр не указан, то подразумевается текущее время. В случае ошибки возвращается false.

Функции для работы с именами файлов:

- string basename(string $path)

Выделяет основное имя файла из пути $path. Примеры:

```
echo basename("/home/somebody/somefile.txt");
// выводит "somefile.txt"
echo basename("/"); // ничего не выводит
echo basename("/."); // выводит "."
echo basename("/./"); // также выводит "."
```

Функция basename() (и все последующие) не проверяет существование файла. Она берет часть строки после самого правого слэша и возвращает ее. Функция basename() правильно обрабатывает как прямые, так и обратные слэши.

- string dirname(string $path)

Возвращает имя каталога, выделенное из пути $path. Функция умеет обрабатывать нетривиальные ситуации:

```
echo dirname("/home/file.txt"); // выводит "/home"
echo dirname("../file.txt"); // выводит ".."
echo dirname("/file.txt");
// выводит "/" под Unix, "\" под Windows
echo dirname("/"); // то же самое
echo dirname("file.txt"); // выводит "."
```

Если функции dirname() передать «чистое» имя файла, она вернет ".", что означает «текущий каталог».

- string tempnam(string $dir, string $prefix)

Генерирует имя файла в каталоге $dir с префиксом $prefix в имени, причем так, чтобы созданный под этим именем в будущем файл был уникален. Для этого к строке $prefix присоединяется некое случайное число. Например, вызов tempnam("/tmp","temp") может

возвратить что-то типа /tmp/temp3a6b243c. Если такое имя нужно создать в текущем каталоге, нужно передать $dir=".".

- string realpath(string $path)

Функция преобразует относительный путь в $path в абсолютный, т. е. начинающийся от корня. Например:

```
echo realpath("../t.php");
// абсолютный путь – например, /home/test.php
echo realpath("."); // выводит имя текущего каталога
```

Файл, который указывается в параметре $path, должен существовать, иначе функция возвращает false.

Функции манипулирования целыми файлами:

- bool copy(string $src, string $dst)

Копирует файл с именем $src в файл с именем $dst. Если файл $dst на момент вызова существовал, осуществляется его перезапись. Функция возвращает true, если копирование прошло успешно, а в случае провала – false.

- bool rename(string $oldname, string $newname)

Переименовывает (или перемещает) файл с именем $oldname в файл с именем $newname. Если файл $newname уже существует, функция возвращает false. В случве успеха, возвращается true.

- bool unlink(string $filename)

Удаляет файл с именем $filename. В случае неудачи возвращает false, иначе – true.

- list File(string $filename)

Считывает файл с именем $filename целиком и возвращает массив-список, каждый элемент которого соответствует строке в прочитанном файле. Функция работает очень быстро – гораздо быстрее использование fopen() и чтение файла по одной строке. Неудобство этой функции состоит в том, что символы конца строки (обычно \n), не вырезаются из строк файла, а также не транслируются, как это делается для текстовых файлов.

- array get_meta_tags(string $filename, int $use_include_path=false);

Функция открывает файл и ищет в нем все тэги <meta> до тех пор, пока не встретится закрывающий тэг </head>. Если очередной тэг <meta> имеет вид:

```
<meta name="название" content="содержимое">
```

то пара название=>содержимое добавляется в результирующий массив, который под и возвращается. Функцию удобно использовать для быстрого получения всех метатегов из указанного файла. Если необязательный параметр $use_include_path установлен, то поиск файла осуществляется не только в текущем каталоге, но и во всех тех, которые назначены для поиска инструкциями include и require.

Другие функции:

- bool ftruncate(int $f, int $newsize)

Функция усекает открытый файл $f до размера $newsize. Файл должен быть открыт в режиме, разрешающем запись. Пример:

```
ftruncate($f,0); // очистить содержимое файла
```

- void fflush(int $f)

Функция заставляет PHP немедленно записать на диск все изменения, которые производились до этого с открытым файлом $f. Для повышения производительности все операции записи в файл буферизируются: например, вызов fputs($f,"Это строка!") не приводит к непосредственной записи данных на диск – сначала они попадают во внутренний буфер (обычно размером 8K). Как только буфер заполняется, его содержимое отправляется на диск, а сам он очищается, и все повторяется вновь. Особенный выигрыш от буферизации чувствуется в сетевых операциях, когда просто глупо отправлять данные маленькими порциями. Функция fflush() вызывается неявно и при закрытии файла.

- int set_file_buffer(int $f, int $size)

Функция устанавливает размер буфера для указанного открытого файла $f. Чаще всего она используется так:

```
set_file_buffer($f,0);
```

Приведенный код отключает буферизацию для указанного файла, так что теперь все данные, записываемые в файл, немедленно отправляются на диск или в сеть.

10.2 Блокирование файла

При интенсивном обмене данными с файлами в мультизадачных операционных системах встает вопрос синхронизации операций чтения/записи между процессами. Например, пусть есть несколько «процессов-писателей» и один «процесс-читатель». Необходимо, чтобы в единицу времени к файлу имел доступ лишь один процесс-писатель, а остальные на этот момент времени как бы «подвисали», ожидая своей очереди. Это нужно, например, чтобы данные от нескольких процессов не перемешивались в файле, а следовали блок за блоком. Для этого используется функция flock(), которая устанавливает так называемую «рекомендательную блокировку» для файла. Это означает, что блокирование доступа осуществляется не на уровне ядра системы, а на уровне программы.

Процессы, которые пользуются рекомендательной блокировкой, будут работать с разделяемым файлом правильно, а остальные… как-нибудь да будут, пока не произойдет «столкновение».

«Жесткая блокировка» (которая в PHP не поддерживается) подобна шлагбауму: никто не сможет проехать, пока его не поднимут.

Для управления блокировками исползуется функция `flock()`.

- `bool flock(int $f, int $operation [, int& $wouldblock])`

Функция устанавливает для указанного открытого дескриптора файла `$f` режим блокировки для текущего процесса. Этот режим задается аргументом `$operation` и может быть одной из следующих констант:

`LOCK_SH` (или 1) – разделяемая блокировка;

`LOCK_EX` (или 2) – исключительная блокировка;

`LOCK_UN` (или 3) – снять блокировку;

`LOCK_NB` (или 4) – эту константу нужно прибавить к одной из предыдущих, если необходимо, чтобы программа не «подвисала» на `flock()` в ожидании своей очереди, а сразу возвращала управление.

В случае, если был затребован режим без ожидания, и блокировка не была успешно установлена, в необязательный параметр-переменную `$wouldblock` будет записано значение истина `true`. В случае ошибки функция возвращает `false`, а в случае успешного завершения – `true`.

Исключительная блокировка. Каждый процесс-писатель в тот момент, когда он уже почти готов начать писать должен быть единственным, кому разрешена запись в файл. Он хочет стать исключительным. Вызвав функцию `flock($f,LOCK_EX)`, можно быть абсолютно уверен, что все остальные процессы не начнут без разрешения писать в файл, пока процесс-писатель не выполнит все свои действия и не вызовет `flock($f, LOCK_UN)` или не закроет файл.

Если в данный момент процесс не единственный претендент на запись, операционная система просто *не выпустит* его из «внутренностей» функции `flock()`, т. е. не допустит его продолжения, пока процесс-писатель не станет единственным. В тот момент, когда процесс, использующий исключительную блокировку, становится активным, все остальные процессы-писатели ожидают (все в той же функции `flock()`), когда он закончит свою работу с файлом. Как только это произойдет, операционная система выберет следующий исключительный процесс, и т. д.

Пример модели процесса с исключительной блокировкой:

```
<?
// инициализация
$f=fopen($f,"a+") or die("Ошибка открытия файла!");
flock($f,LOCK_EX);
// ждем, пока процесс не станет единственным
// только эта программа работает с файлом
fflush($f); // записываем все изменения на диск
flock($f,LOCK_UN); // больше не будем работать с файлом
fclose($f);
// Завершение
?>
```


Если необходимо стирать содержимое файла, ***нельзя*** использовать режим открытия `w` – нужно применять `a+` и функцию `ftruncate()`.

Пример:

```
$f=fopen($f,"a+") or die("Ошибка открытия файла!");
flock($f,LOCK_EX);
// ждем, пока процесс не станет единственным
ftruncate($f,0); // очищаем все содержимое файла
```

Перед тем, как разблокировать файл, нужно использовать функцию `fflush()`, потому что отключение блокировки не ведет к сбросу внутреннего файлового буфера, т. е. некоторые изменения могут быть «сброшены» в файл уже *после* того, как блокировка будет снята.

Исключительная блокировка устанавливается, когда нужно изменять файл. Всегда необходимо использовать при этом режим открытия `r`, `r+` или `a+`. Нельзя применять режим `w`. Перед снятием блокировки нужно вызвать функцию `fflush()`.

Разделяемая блокировка. При использовании исключительной блокировки данные из нескольких процессов-писателей не будут перемешиваться, но как быть с читателями? А вдруг процесс-читатель захочет прочитать как раз из того места, куда пишет процесс-писатель? В этом случае он получит «половинчатые» данные.

Существуют два метода обхода этой проблемы. Первый – это использовать исключительную блокировку. Ведь функция `flock()` не знает, что будет выполнено с файлом, для которого она вызвана. Однако этот метод неудачен, потому что процессов-читателей может быть много, а писателей – мало, и к тому же писатели еще и вызываются, скажем, раз в пару минут, а не постоянно, как читатели. В случае использования исключительной блокировки появится много стоящих в очереди процессов-читателей. Но ведь никакой «аварии» не случится, если один и тот же файл будут читать сразу все процессы-читатели, т.к. чтение из файла его не изменяет. Предоставив исключительную блокировку для читателей, получаем проблемы с производительностью, пе-

рерастающие в катастрофу, когда процессов-читателей становится больше некоторого определенного порога.

Второй (и лучший) способ подразумевает использование разделяемой блокировки. Процесс, который устанавливает этот вид блокировки, будет приостановлен только в одном случае: когда активен другой процесс, установивший исключительную блокировку. Процессы-читатели будут «поставлены в очередь» только тогда, когда активизируется процесс-писатель.

Процесс-писатель должен дождаться, пока читатель не закончит работу. Вызов `flock($f,LOCK_EX)` обязан подождать, пока активна хотя бы одна разделяемая блокировка.

Замечание. Если почти всегда активна разделяемая блокировка, операционная система все равно так распределяет кванты времени, что в некоторые из них можно «включить» исключительную блокировку.

Пример модели процесса с разделяемой блокировкой:

```
<?
// инициализация
$f=fopen($f,"r") or die("Ошибка открытия файла!");
flock($f,LOCK_SH); // ждем, когда процессы-писатели угомонятся
// Пока программа работает с файлом, ни одна другая
// программа в него не пишет
flock($f,LOCK_UN);
// говорим, что мы больше не будем работать с файлом
fclose($f);
// Завершение
?>
```

Разделяемую блокировку необходимо устанавливать, когда нужно только читать из файла, не изменяя его, при этом использовать режим открытия `r`, и никакой другой.

Блокировки с запретом «подвисания». Ко второму параметру функции `flock()` можно прибавить константу `LOCK_NB` для того, чтобы функция не ожидала, когда программа может «двинуться в путь», а сразу же возвращала управление в основную программу. Это может пригодиться, если необходимо, чтобы сценарий бесполезно простаивал, ожидая, пока ему разрешат обратиться к файлу.

Пример использования исключительной блокировки с `LOCK_NB`:

```
$f=fopen("file.txt","a+");
while(!flock($f,LOCK_EX+LOCK_NB)) {
echo "Пытаемся получить доступ к файлу <br>";
sleep(1); // ждем 1 секунду
}
// Работаем
```


Выход из `flock()` может произойти либо в результате отказа блокировки, либо после того, как блокировка будет установлена – но не до того! Таким образом, когда дождемся разрешения доступа к файлу и произойдет выход из цикла `while`, будем иметь исключительную блокировку, закрепленную за файлом.

Использование блокировки при создании счетчика. Без блокировки файла не обойтись при написании сценария счетчика, который бы при каждом своем запуске увеличивал число, хранящееся в файле, и выводил его в браузер. При большой посещаемости сервера могут быть запущены сразу несколько процессов-счетчиков, которые попытаются обратиться к одному и тому же файлу. Если не принять мер, это приведет к тому, что счетчик рано или поздно «обнулится».

Пример простейшего текстового счетчик

```
<?
$f=fopen("counter.dat","a+");
flock($f,LOCK_EX); // дальше будем работать только мы
$count=fread($f,100); // Читаем значение, сохраненное в файле
@$count=$count+1; // Увеличиваем его на 1 (пустая строка = 0)
ftruncate($f,0); // Стираем файл
fwrite($f,$count); // Записываем новое значение
fflush($f); // Сбрасываем файловый буфер
flock($f,LOCK_UN); // Отключаемся от блокировки
fclose($f); // Закрываем файл
echo $count; // Печатаем величину счетчика
?>
```

Здесь применяется только исключительная блокировку, потому что когда нужно вывести на экран счетчик, его нужно увеличить.

10.3 Функции работы с каталогами

С точки зрения операционной системы каталоги – это те же самые файлы, только со специальным именем. То есть директорию можно представить себе как файл, в котором хранятся имена и местоположения других файлов и каталогов. С каждым процессом (в частности, и с работающим сценарием) ассоциирован свой так называемый *текущий каталог*. Все действия по работе с файлами и каталогами осуществляются по умолчанию именно в нем. Например, если открываем файл, указав только его имя, PHP будет искать этот файл именно в текущем каталоге.

Функции для работы с каталогами:

- `bool mkdir(string $name, int $perms)`

Создает каталог с именем `$name` и правами доступа `$perms`. Права доступа для каталогов указываются точно так же, как и для файлов.

Чаще всего значение $perms устанавливают равным 0770 (предваряющий ноль обязателен – он указывает PHP на то, что это – восьмеричная константа, а не десятичное число). *Пример*:

```
mkdir("my_directory",0755);
// создает подкаталог в текущем каталоге
mkdir("/data"); // создает подкаталог data в корневом каталоге
```

В случае успеха функция возвращает true, иначе – false. Пользователь не может создать подкаталог в родительском каталоге, права на запись в который у него отсутствуют.

Атрибуты доступа 0770 означают «доступен для чтения, записи и исполнения для владельца и его группы». Право на «исполнение» показывает, что пользователь сможет *просмотреть* содержимое каталога.

- `bool rmdir(string $name)`

Удаляет каталог с именем $name. В случае успеха возвращает true, иначе – false.

- `bool chdir(string $path)`

Сменяет текущий каталог на указанный. Если такого каталога не существует, возвращает false. Параметр $path может определять и относительный путь, задающийся от текущего каталога. *Примеры*:

```
chdir("/tmp/data"); // переходим по абсолютному пути
chdir("./somathing"); // переход в подкаталог текущего каталога
chdir("something"); // то же самое
chdir(".."); // переходим в родительский каталог
```

Возвращает полный путь к текущему каталогу, начиная от «корня» (/). Если такой путь не может быть отслежен, возвращает false.

Работа с записями. Описываются функции, которые позволяют узнать, какие объекты находятся в указанном каталоге. Механизм работы этих функций базируется на тех же принципах, что и применяемых для файловых операций: сначала каталог открывается, затем из него производится считывание записей, и под конец каталог нужно закрыть.

- `int opendir(string $path)`

Открывает каталог $path для дальнейшего считывания из него информации о файлах и подкаталогах и возвращает его идентификатор. Дальнейшие вызовы readdir() с идентификатором в параметрах будут обращены именно к этому каталогу. Функция возвращает false, если произошла ошибка.

- `string readdir(int $handle)`

Считывает очередное имя файла или подкаталога из открытого ранее каталога с идентификатором $handle и возвращает его в виде строки. Порядок следования файлов в каталоге зависит от операционной системы. Вместе с именами подкаталогов и файлов будут также

получены два специальных элемента: это (ссылка на текущий каталог) и .. (ссылка на родительский каталог). В подавляющем большинстве случаев нам нужно их игнорировать. Если в каталоге все файлы уже считаны, функция возвращает ложное значение. *Пример*:

```
$d=opendir("somewhere");
while(($e=readdir($d))!==false) {...}
```

Оператор !== позволяет точно проверить, была ли возвращена величина false.

- `void closedir(int $handle)`

Закрывает ранее открытый каталог с идентификатором $handle. Можно не закрывать каталоги, т. к. это делается автоматически при завершении программы, однако лучше все-таки это сделать.

- `void rewinddir(int $handle)`

Функция «перематывает» внутренний указатель открытого каталога на начало. После этого можно воспользоваться readdir(), чтобы заново начать считывать содержимое каталога.

10.4 Запуск внешних программ

- `string system(string $command [,int& return_var])`

Функция запускает внешнюю программу, имя которой передано первым параметром, и выводит результат работы программы в выходной поток, т. е. в браузер. Если функции передан также второй параметр – переменная (именно переменная, а не константа!), то в нее помещается код возврата вызванного процесса. Это требует от PHP ожидания завершения запущенной программы. Выходной поток данных программы направляется в браузер. Если нужно этого избежать, воспользуйтесь функциями popen() или exec(). Если же вы, наоборот, нужно, чтобы выходные данные запущенной программы попали прямиком в браузер и никак при этом не исказились (например, вызывается программа, выводящая в стандартный выходной поток какой-нибудь GIF-рисунок), в этом случае в самый раз будет функция PassThru().

- `string exec(string $command [,list& $array] [,int& $return_var])`

Функция запускает указанную программу или команду, но ничего не выводит в браузер. Вместо этого функция возвращает последнюю строку из выходного потока запущенной программы и, если задан параметр $array (который обязательно должен быть переменной), то он заполняется списком строк в выходном потоке – по одной строке на элемент. Если массив уже содержал какие-то данные перед вызовом exec(), новые строки просто добавляются в его конец, а не заменяют старое содержимое массива. При задании параметра-переменной

`$return_var` код возврата запущенного процесса будет помещен в эту переменную. Функция `exec()` тоже дожидается окончания работы нового процесса и только потом возвращает управление в PHP-программу.

- `string EscapeShellCmd(string $command)`

Нельзя допускать возможности передачи данных из браузера пользователя (например, из формы) в функции `system()` и `exec()`. Если это все же нужно сделать, то данные должны быть соответствующим способом обработаны: например, можно защитить все специальные символы обратными слэшами, и т. д. Это и делает функция `EscapeShellCmd()`. Чаще всего ее применяют в таком контексте:

```
system("cd ".EscapeShellCmd($to_directory));
```

Здесь переменная `$to_directory` пришла от пользователя – например, из формы или Cookies.

- `string PassThru(string $command [,int& $return_var])`

Функция запускает указанный в ее первом параметре процесс и весь его вывод направляет прямо в браузер пользователя, один-в-один. Она применяется, если воспользоваться какой-нибудь утилитой для генерации изображений «на лету», оформленной в виде отдельной программы.

Пример:

```
Header("Content-type: image/jpeg");
PassThru("cat my_image.jpg");
```

Функция `Header()` сообщает браузеру пользователя, что данные поступят в графическом формате JPEG, а последующий вызов утилиты `cat` с параметром – именем файла с рисунком – заставляет систему «распечатать» файл в браузер пользователя.

Тема 11 Работа с датами и временем, посылка писем через PHP

11.1 Представление времени в формате timestamp
11.2 Работа с датами, григорианский календарь
11.3 Посылка писем через PHP

11.1 Представление времени в формате timestamp

- `int time()`

Возвращает время в секундах, прошедшее с полуночи 1 января 1970 года по Гринвичу до настоящего момента. Почти все функции по работе со временем имеют дело именно с таким его представлением (которое называется *timestamp*).

- `string microtime()`

Возвращает строку в формате: "`микросекунды секунды`", где секунды – то, что возвращается функцией `time()`, а микросекунды – дробная часть секунд, служащая для более точного измерения промежутков времени.

- `int mktime([int $hour] [,int $minute] [,int $second] [,int $month] [,int $day] [,int $year])`

Функция возвращает значение timestamp, соответствующее указанной дате. Все ее параметры необязательны, но пропускать их можно только справа налево. Если какие-то параметры не заданы, на их место подставляются значения, соответствующие текущей дате. Правильность даты, переданной в параметрах, не проверяется. В случае некорректной даты функция формирует соответствующий timestamp.

Пример:

```
echo date("M-d-Y", mktime(0,0,0,1,1,1998));
// правильная дата
echo date("M-d-Y", mktime(0,0,0,12,32,1997));
// неправильная дата
echo date("M-d-Y", mktime(0,0,0,13,1,1997));
// неправильная дата
```

В результате выводятся три одинаковых числа.

11.2 Работа с датами, григорианский календарь

- `string date(string $format [,int $timestamp])`

Функция возвращает строку, отформатированную в соответствии с параметром `$format` и сформированную на основе параметра `$timestamp` (если последний не задан – то на основе текущей даты). Строка формата может содержать обычный текст, перемежаемый од-

ним или несколькими символами форматирования: U – количество секунд, прошедших с полуночи 1 января 1970 года; z – номер дня от начала года; Y – год, 4 цифры; y – год, 2 цифры; F – название месяца, например, January (формат выдачи месяца зависит от текущих настроек локали); m – номер месяца; M – название месяца, трехсимвольная аббревиатура, например, Jan; d – номер дня в месяце, всегда 2 цифры (первая может быть 0); j – номер дня в месяце без предваряющего нуля; w – день недели, 0 соответствует воскресенью, 1 – понедельнику, и т. д.; l – день недели, текстовое полное название, например, Friday; D – день недели, английское трехсимвольное сокращение, например, Fri; a – am или pm; A – AM или PM; h – часы, 12-часовой формат; H – часы, 24-часовой формат; i – минуты; s – секунды; S – английский числовой суффикс (nd, th и т. д.).

Те символы, которые не были распознаны как форматирующие, подставляются в результирующую строку «как есть». Пример:

```
echo date("l dS of F Y h:i:s A");
echo date("Сегодня d.m.Y");
echo date("Дата файла d.m.Y",filectime("myfile"));
```

- `int checkdate(int $month, int $day, int $year)`

Функция проверяет, существует ли дата, переданная ей в параметрах: вначале ищется месяц, затем – день, и, наконец, – год. Функция проверяет следующее: год должен быть между 1900 и 32 767 включительно; месяц обязан принадлежать диапазону от 1 до 12; число должно быть допустимым для указанного месяца и года (если год високосный).

- `array getdate(int $timestamp)`

Возвращает ассоциативный массив, содержащий данные об указанном времени. В массив будут помещены следующие ключи и значения: seconds – секунды; minutes – минуты; hours – часы; mday – число; wday – день недели, число; mon – номер месяца; year – год; yday – номер дня с начала года; weekday – полное название дня недели, например, Friday; month – полное название месяца, например, January.

Григорианский календарь – это тот календарь, который мы постоянно используем в своей жизни. В России он был введен Петром I в 1700 году. Каждой дате соответствует свой Julian Day Count (JDC). JDC – это число дней, прошедших с определенной даты (где-то с 4714-го года до нашей эры). Это нужно например, чтобы вычислить количество дней между двумя датами в формате "дд.мм.гггг". Обе даты переводятся в в JDC и определяется разность получившихся величин.

- `int GregorianToJD(int $month, int $day,`

`int $year)`

Преобразует дату в формат JDC. Допустимые значения года для григорианского календаря – от 4714 года до нашей эры до 9999 года нашей эры.

- `string JDToGregorian(int $julianday)`

Преобразует дату в формате JDC в строку месяц/число/год. Чтобы разбить эту строку на составляющие можно воспользоваться функцией explode():

```
$jd = GregorianToJD(10,11,1970);
echo "$jd<br>\n";
$gregorian = JDToGregorian($jd);
echo "$gregorian<br>\n";
$list=explode($gregorian,"/");
```

- `mixed JDDayOfWeek(int $julianday, int $mode)`

Функция JDDayOfWeek() возвращает день недели, на который приходится указанная JDC-дата. Параметр $mode задает, в каком виде должен быть возвращен результат: 0 – номер дня недели (0 — воскресенье, 1 — понедельник, и т. д.); 1 – английское название дня недели; 2 – сокращение английского названия дня недели.

11.3 Посылка писем через PHP

Одно из самых мощных средств PHP – возможность автоматической посылки писем по электронной почте, минуя использование внешних программ и утилит. Функция отсылки встроена в PHP.

Функция отправки письма

- `bool mail(string $to, string $subject, string $msg [,string $headers])`

Функция mail() посылает сообщение с телом $msg (это может быть «многострочная строка», т. е. переменная, содержащая несколько строк, разделенных символом перевода строки) по адресу $to. Можно задать сразу нескольких получателей, разделив их адреса пробелами в параметре $to. *Пример*:

```
mail("rasmus@lerdorf.on.ca ca.ok@oklab.ru,
"My Subject", "Line 1\nLine 2\nLine 3");
```

Если указан четвертый параметр, переданная в нем строка вставляется между концом стандартных почтовых заголовков (таких как To, Content-type и т. д.) и началом текста письма. Этот параметр используется для задания дополнительных заголовков письма. *Пример*:

```
mail("ssb@guardian.no dk@dizain.ru",
"the subject",
"Line 1\nLine 2\nLine 3",
"From: webmaster@$SERVER_NAME\n".
```

```
"Reply-To: webmaster@$SERVER_NAME\n".
"X-Mailer: PHP/" . phpversion());
```

Гораздо лучше было бы включить указанные заголовки прямо в тело письма `$msg` (в начало тела), отделив их от самого письма пустой строкой (как в стандарте HTTP). То же самое применимо и к параметру `$subject`: лучше задавать в нем всегда пустую строку и указывать заголовок `Subject` в самом письме.

Посылка в указанной кодировке. Чтобы не возникало приблем с кодировками при отправке писем, нужно функции `mail()` передавать только адрес получателя и текст письма. Ни заголовков, ни темы – и то и другое должно присутствовать в самом письме. *Пример*:

```
$message=
"From: Лист рассылки
To: Иванов Иван Иванович
Subject: Пробная рассылка
Content-type: text/plain; charset=windows-1825
Уважаемый товарищ! Всего хорошего!";
Mail("ivanov@ivan.ivanovich.ru","",$message);
```

Заголовок `Content-type` (в некоторых системах он обязательно должен стоять последним) задает, что, во-первых, письмо доставляется как простой текст (`text/plain`), а во-вторых, что его кодировка – Windows. Тело письма отделяется от заголовков пустой строкой, с тем, чтобы почтовая программа могла понять, где кончаются заголовки и начинается тело.

Тема 12 Работа с WWW

12.1 Установка заголовков ответа
12.2 Работа с Cookies

12.1 Установка заголовков ответа

Рассмотрим средства PHP для работы с заголовками HTTP.

Вывод заголовка

- `int Header(string $string)`

Функция предназначена для установки заголовков ответа, которые будут переданы браузеру – по одному заголовку на вызов. Она может быть вызвана только до первой команды вывода сценария (если до этого не использовалась функция буферизации `ob_start()`). Текст вне `<?` и `?>` также рассматривается как оператор вывода, потому не нужно делать лишних пробелов до первой «скобки» `<?` в сценарии (и в особенности в файле, который этим сценарием включается) и, конечно, после последнего ограничителя `?>` во включаемом файле. *Пример*:

```
// перенаправляет браузер на сайт PHP
Header("Location: http://www.php.net");
// принудительно завершаем сценарий
exit;
```

Запрет кэширования. Еще одно полезное приложение функции `Header()` – запрет кэширования документа браузером и Proxy-серверами. Большинство сценариев формируют документы, которые при каждом запуске программы изменяются. Выход – использовать в начале сценария следующие команды:

```
Header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
// Дата в прошлом
Header("Cache-Control: no-cache, must-revalidate");
// HTTP/1.1
Header("Pragma: no-cache"); // HTTP/1.0
Header("Last-Modified:"
.gmdate("D,dMYH:i:s")."GMT");
```

Для полного запрета кэширования нужно всегда посылать 4 указанных заголовка, и ни один пропустить нельзя – в противном случае не сработает либо браузер, либо Proxy-сервер.

Получение заголовков запроса. Для получения всех заголовков запроса следует воспользоваться функцией `GetAllHeaders()`:

- `array GetAllHeaders()`

Функция `GetAllHeaders()` возвращает ассоциативный массив, содержащий данные о HTTP-заголовках запроса клиента, породившего

запуск сценария. Ключи массива содержат названия заголовков, а значения – их величины. Пример:

```
$headers = GetAllHeaders();
foreach($headers as $header=>$value)
echo "$header: $value<br>\n";
```

Функция `GetAllHeaders()` поддерживается PHP только в том случае, если он установлен в виде модуля Apache.

12.2 Работа с Cookies

Термин *Cookies* (это множественное число, произносится как «кукис» или «куки»). В буквальном переводе слово звучит как «печенье», и почему компания Netscape так назвала свое изобретение, неизвестно. Слово Cookies применяется с большой буквы, во множественном числе и мужского рода. В единственном числе это понятие записывается Cookie и произносится на русский манер – «кука».

Cookie – это небольшая именованная порция информации, которая может сохраняться в настройках браузера пользователя между сеансами. Причина, по которой применяются Cookies – большое количество посетителей сервера, а также нежелание иметь нечто подобное базе данных для хранения информации о каждом посетителе. Поиск в такой базе может очень и очень затянуться, и, в то же время, нет никакого смысла централизованно хранить столь отрывочные сведения. Использование Cookies фактически перекладывает задачу на плечи браузера, решая одним махом как проблему быстродействия, так и проблему большого объема базы данных с информацией о пользователе.

Самый распространенный пример применения Cookies – логин и пароль пользователя, использующего защищенные ресурсы сайта. Эти данные, между открытиями страниц хранятся в Cookies, для того чтобы пользователю не пришлось их каждый раз набирать заново.

У каждого Cookie есть определенное время жизни, по истечение которого он автоматически уничтожается. Существуют также Cookies, которые «живут» только в течение текущего сеанса работы с браузером (это могут быть, например, имя и пароль, введенные при авторизации), или же идентификатор сессии.

Каждый Cookie устанавливается сценарием на сервере. Сценарий также получает все Cookies, которые сохранены на удаленном компьютере, при каждом своем запуске, так что он может в любой момент времени узнать, что же у пользователя установлено. Для этого он должен послать браузеру специальный заголовок вида:

```
Set-cookie: данные
```

Кроме этого, имеется также информация об имени сервера, установившего этот Cookie, и URL каталога, в котором находился сценарий-хозяин в момент инициализации. Имя сервера и каталог нужны

для следующего: сценарию передаются только те Cookies, у которых параметры с именем сервера и каталога совпадают соответственно с хостом и каталогом сценария. Поэтому невозможно получить доступ к «чужим» Cookies – браузер просто не будет посылать их серверу.

Однако в PHP этот процесс скрыт за функцией `SetCookie()`.

Недостатки Cookies. Первый – не все браузеры поддерживают Cookies. Второй недостаток заключается в том, что каждый браузер хранит свои Cookies отдельно. То есть Cookies, установленные при пользовании Internet Explorer, не будут «видны» при работе в Netscape, и наоборот.

Установка Cookie. Команда установки Cookie – это просто один из заголовков ответа, передаваемых сервером браузеру. То есть, перед тем как выводить `Content-type`, можно указать некоторые команды для установки Cookie. Формат команды (как и всякий заголовок, записывается она в одну строку):

```
Set-Cookie: name=value; expires=дата;
domain=имя_хоста; path=путь; secure
```

Существует и другой подход активизировать Cookie – при помощи HTML-тэга `<meta>`. *Формат тэга*:

```
<meta http-equiv="Set-Cookie"
      content="name=value; expires=дата;
      domain=имя_хоста; path=путь; secure">
```

Параметры Cookie:

`name` – имя, закрепленное за Cookie. Имя должно быть URL-кодированным текстом, т. е. состоять только из алфавитно-цифровых символов.

`value` – текст, который будет рассматриваться как значение Cookie. “тот текст должен быть URL-кодирован.

`expires` – необязательная пара `expires=дата` задает время жизни Cookie. Cookie самоуничтожится, как только наступит указанная дата. Например, если задать `expires=Friday,31-Dec-09 23:59:59 GMT`, то «печенье» будет «жить» только до 31 декабря 2009 года. Если этот параметр не указан, то временем жизни будет считаться вся текущая сессия работы браузера, до того момента, как пользователь его закроет.

`domain` – параметр `domain=имя_хоста` задает имя хоста, с которого установили Cookie. Его можно менять вручную, прописав нужный адрес, и таким образом «подарить» Cookie другому хосту. Только в том случае, если параметр не задан, имя хоста определяется браузером автоматически.

`path` – параметр `path=путь` описывает каталог (точнее, URI), в котором расположен сценарий, установивший Cookie. Этот параметр

также можно установить вручную, записав в него не только каталог, а вообще все, что угодно. Однако указав хост, отличный от хоста сценария, или путь, отличный от URI каталога (или родительского каталога) сценария, тем самым никогда больше не увидим Cookie в этом сценарии.

`secure` – параметр связан с защищенным протоколом передачи HTTPS. Он используется при написании сценария для проведения банковских операций с кредитными карточками (или иные, требующие повышенной безопасности).

Значения всех параметров Cookie должны быть URL-кодированы.

Получение Cookies из браузера. Все Cookies хранятся в переменной окружения `HTTP_COOKIE` в таком же формате, как и `QUERY_STRING`, только вместо `&` используется `;`. Например, если установили два Cookies: `cookie1=value1` и `cookie2=value2`, то в переменной окружения `HTTP_COOKIE` будет следующее:
`cookie1=value1;cookie2=value2.`

Сценарий должен разобрать эту строку, распаковать ее и затем работать по своему усмотрению.

Установка Cookie в PHP. Так как Cookie фактически представляет собой обычный заголовок, сделать это можно только перед первой командой вывода в сценарии.

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

Вызов `SetCookie()` определяет новый Cookie, который тут же посылается браузеру вместе с остальными заголовками. Все аргументы, кроме имени, необязательны. Если задан только параметр `$name` (имя Cookie), то Cookie с указанным именем у пользователя удаляется. Можно пропускать аргументы, которые не нужно задавать, заменяя их пустыми строками "". Аргументы `$expire` и `$secure`, не могут быть представлены строками, а потому вместо пустых строк здесь нужно использовать `0`. Параметр `$expire` задает timestamp, который, например, может быть сформирован функциями `time()` или `mktime()`. Параметр `$secure` говорит о том, что величина Cookie может передаваться только через безопасное HTTPS-соединение.

Примеры использования `SetCookie()`:

```
// Cookie на одну сессию, т. е. до закрытия браузера
SetCookie("TestCookie","Test Value");
// Эти Cookies уничтожаются браузером через 1 час после установки
SetCookie("TestCookie",$val,time()+3600);
SetCookie("TestCookie",$val,time()+3600,
"/~rasmus/",".utoronto.ca",1);
```


После вызова функции `SetCookie()` только что созданный Cookie сразу появляется среди глобальных переменных как переменная с заданным в параметре `$name` именем. Она появится и при следующем запуске сценария – даже если `SetCookie()` в нем и не будет вызвана. Параметр `$value` автоматически URL-кодируется при посылке на сервер, а при получении Cookie – автоматически декодируется, как это происходит и с данными формы.

Пример: счетчик посещения страницы конкретным посетителем. Запуская данный сценарий, пользователь будет видеть, сколько раз он уже гостил на странице.

Индивидуальный счетчик посещений:

```
if(!isSet($Counter)) $Counter=0;
$Counter++;
SetCookie("Counter",$Counter,0x7FFFFFFF);
echo "Вы запустили этот сценарий $Counter раз!";
```

Если понадобится сохранять в Cookies не только строки, но и сложные объекты, то объект нужно сначала преобразовать в строку (например, при помощи `Serialize()`) и поместить ее в Cookie. А потом, наоборот, распаковать строку, используя `Unserialize()`. Если сохраняемый массив имеет небольшой размер, каждый его элемент можно разместить в отдельном Cookie:

```
SetCookie("Arr[0]","aaa");
SetCookie("Arr[1]","bbb");
SetCookie("Arr[2][0]","ccc"); // многомерный массив
```

По сути, Cookie с именем `Arr[0]` ничем не отличается с точки зрения браузера и сервера от обычного Cookie. Зато PHP, получив Cookie с именем, содержащим квадратные скобки, поймет, что это на самом деле элемент массива, и создаст его (массив).

В большинстве браузеров число Cookies, которые могут быть установлены одним сервером, ограничено, причем ограничено именно их количество, а не суммарный объем. Поэтому лучше будет воспользоваться функцией `Serialize()` и установить один Cookie.

Получение Cookie в PHP. Предположим, сценарий отработал и установил какой-то Cookie, например, с именем `Cook` и значением `Val`. В следующий раз при запуске этого сценария ему передастся пара типа `Cook=Val` (через специальную переменную окружения). PHP это событие перехватит и автоматически создаст переменную `$Cook` со значением `Val`. То есть интерпретатор действует точно так же, как если бы значение Cookie пришло из формы. Та переменная, которую установили в прошлый раз, будет доступна и сейчас!

Авторизация. Часто бывает нужно, чтобы на какой-то URL могли попасть только определенные пользователи. А именно, только те, у ко-

торых есть зарегистрированное имя (*login*) и пароль (*password*). Механизм *авторизации* призван упростить проверку данных таких пользователей. Рассмотрим один из самых простых типов авторизации – *basic-авторизацию*. Предположим, что сценарий посылает браузеру пользователя следующий заголовок:

```
WWW-Authenticate: Basic realm="имя_зоны"
HTTP/1.0 401 Unauthorized"
```

Строка `имя_зоны` задает идентификатор, который будет определять, к каким ресурсам будет разрешен доступ зарегистрированным пользователям. При программировании CGI-сценариев этот параметр используется в основном для формирования приветствия (подсказки) в диалоговом окне, появляющемся в браузере пользователя (там отображается имя зоны).

Затем посылается тело документа. В браузере пользователя появляется диалоговое окно, в котором предлагается вести login и password. После того как пользователь это сделает, управление передается обратно серверу, который среди обычных заголовков запроса получает примерно такой:

```
Authorization: Basic TG9naW46UGFzcw==
```

Это закодированные данные, введенные пользователем. Далее этот заголовок должен передаться сценарию. Сценарий, декодировав его, может решить: то ли повторить всю процедуру сначала (если имя или пароль неправильные), или же начать работать с сообщением «OK, все в порядке, вы – зарегистрированный пользователь».

Предположим, что сценарий подтвердил верность данных и «пропустил» пользователя. В этом случае происходит еще одна вещь: login и password пользователя запоминаются в скрытом Cookie, «живущем» в течение одной сессии работы с браузером. Затем, заголовок

```
Authorization: Basic значение_Cookie
```

будет присылаться для любого сценария (и даже для любого документа) на сервере. Таким образом, посетителю, зарегистрировавшемуся однажды, нет необходимости каждый раз заново набирать свое имя и пароль в течение текущего сеанса работы с браузером, т. е., пока пользователь его не закроет.

После верной авторизации при вызове любого сценария будет установлена переменная окружения `REMOTE_USER`, содержащая имя пользователя.

SSI и функция virtual(). Для одного и того же документа в Apache нельзя применять два «обработчика». Иными словами, действует принцип: либо PHP, либо SSI (Server-Side Includes – включения на стороне сервера). Однако в PHP имеются определенные средства для «эмуляции» SSI-конструкции

- `include virtual`

Конструкция `include virtual` загружает файл, URL которого указан у нее в параметрах, обрабатывает его нужным обработчиком и выводит в браузер. То есть все происходит так, будто указанный URL был затребован виртуальным браузером. Большинство SSI-файлов ограничиваются использованием этой возможности.

- `int virtual(string $url)`

Функция `virtual()` представляет собой процедуру, которая может поддерживаться только в случае, если PHP установлен как модуль Apache. Она делает то же самое, что и SSI-инструкция `<!--#include virtual=...-->`, т.е. она генерирует новый запрос серверу, обрабатываемый им обычным образом, а затем выводит данные в стандартный поток вывода.

Чаще всего функция `virtual()` используется для запуска внешних CGI-сценариев, написанных на другом языке программирования, или же для обработки SSI-файлов более сложной структуры. Для включения обычных PHP-файлов с участками кода функция `virtual()` неприменима – это выполняет оператор `include`.

Тема 13 Управление интерпретатором и сессиями

13.1 Управление интерпретатором
13.2 Управление сессиями

13.1 Управление интерпретатором

PHP имеет множество различных настроечных параметров, которые иногда приходится изменять или проверять.

Информационные функции:

- `int phpinfo()`

Функция выводит в браузер настройки PHP и параметры вызова сценария: версия PHP; опции, которые были установлены при компиляции PHP; информация о дополнительных модулях; переменные окружения, в том числе и установленные сервером при получении запроса от пользователя на вызов сценария; версия операционной системы; состояние основных и локальных настроек интерпретатора; HTTP-заголовки; лицензия PHP. *Пример*:

```
<?
phpinfo();
?>
```

Функция `phpinfo()` в основном применяется при первоначальной установке PHP для проверки его работоспособности.

- `string phpversion()`

Функция возвращает текущую версию PHP.

- `int getlastmod()`

Функция возвращает время последнего изменения файла, содержащего сценарий. Она учитывает время изменения только главного файла, того, который запущен сервером, но не файлов, которые включаются в него директивами `require` или `include`. Время возвращается в формате timestamp (то есть, это число секунд, прошедших с 1 января 1970 года до момента модификации файла), и затем оно может быть преобразовано в читаемую форму. *Пример*:

```
echo "Последнее изменение: ".date("d.m.Y H:i.s.",
getlastmod());
// Выводит 'Последнее изменение: 13.11.2008 11:23.12'
```

Настройка параметров PHP. Все параметры находятся в файле `php.ini`. Задаются они в формате `параметр=значение`, на одной строке может определяться только один параметр. Любые символы, расположенные после ; и до конца строки, игнорируются (точка с запятой – это признак начала комментария). Если PHP установлен как модуль Apache, можно задавать настройки PHP в главном конфигурационном файле сервера `httpd.conf` или в файлах `.htaccess`. Для этого перед именем каждого параметра нужно поставить префикс `php_` и разделять имя параметра и его значение не знаком равенства, а пробелом. *Список настроечных директив PHP:*

- `error_reporting`

Устанавливает уровень строгости для системы контроля ошибок PHP. Значение этого параметра должно быть целым числом, которое интерпретируется как десятичное представление двоичной битовой маски. Установленные в 1 биты задают, насколько детальным должен быть контроль. Можно использовать константы: `1` – `E_ERROR` – фатальные ошибки; `2` – `E_WARNING` – общие предупреждения; `4` – `E_PARSE` – ошибки трансляции; `8` – `E_NOTICE` – предупреждения; `16` – `E_CORE_ERROR` – глобальные предупреждения; `32` – `E_CORE_WARNING` – глобальные ошибки.

Наиболее часто встречающееся сочетание – 7 (1+2+4), которое задает полный контроль, кроме некритичных предупреждений интерпретатора (таких, например, как обращение к неинициализированной переменной). Оно часто задается по умолчанию при установке PHP.

- `magic_quotes_gpc on|off`

Эта настройка указывает PHP, нужно ли ему ставить дополнительный слэш перед всеми апострофами ', кавычками ", обратными слэшами \ и нулевыми символами (`0`) при приеме данных из браузера пользователя – например, поступивших из формы.

- `max_execution_time`

Директива устанавливает время (в секундах), через которое работа сценария будет принудительно прервана. Используется для того, чтобы запретить пользователям захватывать слишком много ресурсов центрального процессора и избежать «зависания» сценария.

- `track_vars on|off`

Если этот параметр установлен в `On`, все данные, доставленные методами `GET` и `POST`, а также Cookies, будут дополнительно помещены в глобальные массивы `$HTTP_GET_VARS`, `$HTTP_POST_VARS` и `$HTTP_COOKIE_VARS` соответственно.

Контроль ошибок. Одна из самых сильных черт PHP – возможность отображения сообщений об ошибках прямо в браузере. В зависимости от состояния интерпретатора сообщения будут либо выводиться в браузер, либо подавляться. Для установки режима вывода ошибок служит функция `Error_Reporting()`.

- `int Error_Reporting([int $level])`

Устанавливает уровень строгости системы контроля ошибок PHP, т. е. величину параметра `error_reporting` в конфигурации PHP.

Оператор отключения ошибок. Если оператор `@` поставить перед любым выражением, то все ошибки, которые там возникнут, будут проигнорированы. *Пример*:

```
if(!@filemtime("notextst.txt"))
echo "Файла не существует!";
```

Если убрать оператор `@` – получите сообщение: «Файл не найден», а только после этого – вывод оператора `echo`. С оператором `@` предупреждение будет подавлено.

Пример использования оператора @. Пусть имеется форма с submit-кнопкой, и нужно в сценарии определить, нажата ли она.

1-ый способ:

```
<?
if(!empty($submit)) echo "Кнопка нажата!";
?>
```

2-ой способ:

```
<?
if(@$submit) echo "Кнопка нажата!"
?>
<form action=<?=$SCRIPT_NAME?> method=post>
<input type=submit name=submit value="Go!">
</form>
```

Принудительное завершение программы

- `void exit()`

Функция завершает работу сценария. Перед окончанием программы вызываются функции-финализаторы.

- `void die(string $message)`

Функция делает почти то же самое, что и `exit()`, только перед завершением работы выводит строку, заданную в параметре `$message`. Ее применяют, если нужно напечатать сообщение об ошибке и аварийно завершить программу. *Пример*:

```
$filename='/path/to/data-file';
$file=fopen($filename, 'r') or die("не могу открыть
файл $filename!");
```

Финализаторы. Разработчики PHP предусмотрели возможность указать в программе функцию-финализатор, которая будет автоматически вызвана, как только работа сценария завершится. В такой функции можно, например, записать информацию в кэш или обновить файл журнала работы программы. Для этого нужно во-первых, написать саму функцию и дать ей любое имя; во-вторых зарегистрировать ее как

финализатор, передав ее имя стандартной функции `Register_shutdown_function()`.

- `int Register_shutdown_function(string $func)`

Регистрирует функцию с указанным именем, чтобы она автоматически вызывалась перед возвратом из сценария. Функция будет вызвана как при окончании программы, так и при вызовах `exit()` или `die()`, а также при фатальных ошибках, приводящих к завершению сценария – например, при синтаксической ошибке.

Можно зарегистрировать несколько финальных функций, которые будут вызываться в том же порядке, в котором они регистрировались. Однако финальная функция вызывается уже после закрытия соединения с браузером клиента.

Генерация кода во время выполнения. Так как PHP является транслирующим интерпретатором, в нем заложены возможности по созданию и выполнению кода программы прямо во время ее выполнения. То есть можно писать сценарии, которые в буквальном смысле создают сами себя, точнее, свой код! Это незаменимо при написании шаблонизаторов и функций, занимающихся динамическим формированием писем.

Выполнение кода

- `int eval(string $code)`

Функция берет параметр `$st` и, рассматривая его как код программы на PHP, запускает. Если этот код возвратил какое-то значение оператором `return`, `eval()` также вернет эту величину. Параметр `$st` представляет собой строку, содержащую участок PHP-программы. То есть в ней может быть все, что допустимо в сценариях: ввод-вывод, в том числе закрытие и открытие тэгов `<?` и `?>`; управляющие инструкции: циклы, условные операторы и т. д.; объявления и вызовы функций; вложенные вызовы функции `eval()`.

При этом нужно помнить несколько важных правил:

1 Код в `$st` будет использовать те же самые глобальные переменные, что и вызвавшая программа. Т.е. переменные *не локализуются* внутри `eval()`.

2 Любая критическая ошибка в коде строки `$st` приведет к завершению работы всего сценария. Это значит, что нельзя перехватить *все* ошибки в коде, вставив его в `eval()`. Синтаксические ошибки и предупреждения, возникающие при трансляции кода в `$st`, не приводят к завершению работы сценария, а лишь вызывают возврат из `eval()` значения ложь.

Нельзя забывать, что переменные в строках, заключенных в двойные кавычки, в PHP интерполируются (то есть заменяются на соответ-

ствующие значения). Нужно применять строки в апострофах для параметра, передаваемого `eval()`. *Пример*:

```
eval("$a=$b;"); // Неверно!
eval("\$a=\$b");// Вы хотели написать так
eval('$a=$b');  // но так будет короче
```

Сила функции `eval()` заключается в том, что параметр `$st` может являться не статической строковой константой, а сгенерированной переменной.

Генерация функций.

- `string create_function(string $args, string $code)`

Создает функцию с уникальным именем, выполняющую действия, заданные в коде `$code` (это строка, содержащая программу на PHP). Созданная функция будет принимать параметры, перечисленные в `$args`. Возвращаемое значение представляет собой уникальное имя функции, которая была сгенерирована. *Примеры*:

```
$Mul=create_function('$a,$b', 'return $a*$b;');
$Neg=create_function('$a', 'return -$a;');
echo $Mul(10,20); // выводит 200
echo $Neg(2); // выводит -2
```

Проверка синтаксической корректности кода. С помощью `create_function()` можно проверить, является ли некоторая строка верным PHP-кодом, не запуская при этом сам код. Если создание функции с телом – заданной строкой – прошло успешно, значит, код синтаксически корректен. *Пример*:

```
$fname="file.php";
$code=join("",File($fname));
if(create_function("","?>$code<?"))
   echo "Файл $fname является программой на PHP";
else
   echo "Файл $fname – не PHP-сценарий";
```

Другие функции:

- `void usleep(int $micro_seconds)`

Вызов этой функции позволяет сценарию «замереть» не указанное время (в микросекундах). Существует также функция `sleep()`, которая принимает в параметрах не микросекунды, а *секунды*, на которые нужно задержать выполнение программы.

- `int uniqid(string $prefix)`

Функция `uniqid()` возвращает строку, при каждом вызове отличающуюся от результата предыдущего вызова. Параметр `$prefix` задает префикс (до 114 символов длиной) этого идентификатора.

13.2 Управление сессиями

Сессии представляют собой механизм, позволяющий хранить некоторые (и произвольные) данные, индивидуальные для каждого пользователя (например, его имя и номер счета), между запусками сценария. Термин «сессия» является транслитерацией от английского слова session, что в буквальном переводе должно бы означать «сеанс». Фактически, *сессия* – это некоторое место долговременной памяти (обычно часть на жестком диске и часть – в Cookies браузера), которое сохраняет свое состояние между вызовами сценариев одним и тем же пользователем. Поместив в сессию переменную (любой структуры), при следующем запуске сценария получим ее в целости и сохранности.

Зачем нужны сессии? В Web-программировании есть класс задач, который может вызвать много проблем, если писать сценарии «в лоб». Речь идет о невозможности запустить программу на длительное время, позволив ей при этом обмениваться данными с пользователями.

Сценарии должны запускаться, моментально выполняться и возвращать управление системе. Пусть нужно написать форму, содержащую большое число полей, которую было бы глупо поместить на одну страницу. Нужно разбить процесс заполнения формы на несколько этапов, и представить их в виде отдельных HTML-документов. В первом документе с диалогом у пользователя может запрашиваться его имя и фамилия, во втором (если первый был заполнен верно) – данные о его месте жительства, и в третьем – номер кредитной карточки.

В любой момент можно вернуться на шаг назад, чтобы исправить те или иные данные. Если все в порядке, накопленная информация обрабатывается – например, помещается в базу данных.

Реализация такой схемы оказывается для Web-приложений довольно нетривиальной проблемой. Действительно, придется хранить все ранее введенные данные в каком-нибудь временном хранилище, которое должно аннулироваться, если пользователь вдруг передумает и «уйдет» с сайта. Для этого можно использовать функции сериализации и файлы. Однако ими проблема решается только наполовину: нужно также привязывать конкретного пользователя к конкретному временному хранилищу. Эти проблемы решаются с применением сессий PHP.

Механизм работы сессий. Должен существовать механизм, который бы позволил PHP идентифицировать каждого пользователя, запустившего сценарий. То есть при следующем запуске PHP нужно однозначно определить, кто его запустил: тот же человек, или другой. Делается это путем присвоения клиенту так называемого уникального *идентификатора сессии*. Чтобы этот идентификатор был доступен при каждом запуске сценария, PHP помещает его в Cookies браузера. Использовать Cookies не обязательно, существует и другой способ.

Зная идентификатор (будем называть его SID), PHP может определить, в каком же файле на диске хранятся данные пользователя. Чтобы сохранять переменную (обязательно глобальную) в сессии нужно ее зарегистрировать с помощью специальной функции. После регистрации при следующем запуске сценария *тем же* пользователем она получит то же самое значение, которое было у нее при предыдущем завершении программы. Это произойдет потому, что при завершении сценария PHP автоматически сохраняет все переменные, зарегистрированные в сессии, во временном хранилище. Можно в любой момент аннулировать переменную – «вычеркнуть» ее из сессии, или же уничтожить все данные сессии. Чтобы задать промежуточное хранилище, которое использует PHP, можно написать соответствующие функции и зарегистрировать их как *обработчики сессии*. Однако В PHP существуют обработчики по умолчанию, которые хранят данные в файлах.

Инициализация сессии. Прежде, чем работать с сессией, ее необходимо *инициализировать*. Делается это путем вызова специальной функции `session_start()`.

- `void session_start()`

Функция инициализирует механизм сессий для текущего пользователя, запустившего сценарий. По ходу инициализации она выполняет ряд действий:

1. Если посетитель запускает программу впервые, у него устанавливается Cookies с уникальным идентификатором, и создается временное хранилище, ассоциированное с этим идентификатором.
2. Определяется, какое хранилище связано с текущим идентификатором пользователя.
3. Если в хранилище имеются какие-то переменные, их значения восстанавливаются (создаются глобальные переменные, которые были сохранены в сессии при предыдущем завершении сценария).

Регистрация переменных. Чтобы сохранить переменную в сессии, ее предварительно нужно зарегистрировать. Для этого предназначена функция `session_register()`.

- `bool session_register(mixed $name [, mixed $name1, ...])`

Функциця принимает в параметрах одно или несколько имен переменных (имена задаются в строках, без знака $ слева), регистрирует их в текущей запущенной сессии и возвращает значение «истина», если все прошло корректно. В функцию можно передавать не одну строку в каждом параметре, а сразу список строк. Каждая такая строка будет регистрировать отдельную переменную с соответствующим именем. Элементом списка может быть список строк, и т. д. Нет ничего страшного, если дважды зарегистрировать одну и ту же переменную в сессии. Так и происходит при повторном запуске сценария.

Пример работы с сессиями:

```
<?
session_start();
session_register("count");
$count=@$count+1;
?>
<body>
<h2>Счетчик</h2>
В текущей сессии работы с браузером Вы открыли эту
страницу <?=$count?> раз(а). Закройте браузер, что-
бы обнулить счетчик.
</body>
```

Идентификатор сессии и имя группы. На одном и том же сайте могут сосуществовать сразу несколько сценариев, которые нуждаются в услугах поддержки сессий PHP. Они «ничего не знают» друг о друге, поэтому временные хранилища для сессий должны выбираться не только на основе идентификатора пользователя, но и на основе того, какой из сценариев запросил обслуживание сессии.

Имя группы сессий. Пусть разработчик **A** написал сценарий счетчика. Он использует переменную `$count`, и не имеет никаких проблем, до тех пор, пока разработчик **B**, ничего не знающий о сценарии **A**, не создал систему статистики, которая тоже использует сессии. Он также регистрирует переменную `$count`, не зная о том, что она уже «занята». В результате пользователь, запустив сначала сценарий разработчика **B**, а потом – **A**, видит, что данные счетчиков перемешались.

Нужно разграничить сессии, принадлежащие одному сценарию, от сессий, принадлежащих другому. Для этого используются группы сессий. Можно давать *группам сессий* непересекающиеся имена, и сценарий, знающий имя своей группы сессии, сможет получить к ней доступ. Теперь разработчики **A** и **B** могут оградить свои сценарии от проблем с пересечениями имен переменных. Достаточно в первой программе указать PHP, что мы хотим использовать группу с именем, скажем, `sesA`, а во второй – `sesB`.

- `string session_name([string $newname])`

Функция устанавливает или возвращает имя группы сессии, которая будет использоваться PHP для хранения зарегистрированных переменных. Если `$newname` не задан, то возвращается текущее имя. Если же этот параметр указан, то имя группы будет изменено на `$newname`, при этом функция вернет предыдущее имя.

`Session_name()` лишь сменяет имя текущей группы и сессии, но не создает новую сессию и временное хранилище! Нужно в большинстве случаев вызывать `session_name` (имя_группы) еще *до* ее инициализации – вызова `session_start()`.

Если функция `session_name()` не была вызвана до инициализации, PHP будет использовать имя по умолчанию – `PHPSESID`.

Имя группы сессий, устанавливаемое рассматриваемой функцией, – это имя Cookie, который посылается в браузер клиента для его идентификации. Таким образом, пользователь может одновременно активизировать две и более сессий – с точки зрения PHP он будет выглядеть как два или более различных пользователя. Следует помнить, что, случайно установив в сценарии Cookie, имя которого совпадает с одним из имен группы сессий, Cookie «затрется». *Пример*:

```
<?
session_name("CounterScript"
session_start();
session_register("count");
$count=@$count+1;
?>
В текущей сессии Вы открыли эту страницу
<?=$count?> раз(а).
```

Лучше всегда указывать имя группы сессии вручную, не полагаясь на значение по умолчанию.

Идентификатор сессии. Идентификатор сессии (`SID`) фактически является именем временного хранилища, которое будет использовано для хранения данных сессии между запусками сценария. Итак, один `SID` – одно хранилище. Нет `SID`, нет и хранилища, и наоборот.

Как же соотносится идентификатор сессии и имя группы? Имя – это *собирательное название* для нескольких сессий (то есть, для многих `SID`), запущенных разными пользователями. Один и тот же клиент *никогда* не будет иметь два различных `SID` в пределах одного имени группы. Но его браузер вполне может работать с несколькими `SID`, расположенными логически в разных «пространствах имен».

Итак, все `SID` уникальны и однозначно определяют сессию на компьютере, выполняющем сценарий – независимо от имени сессии. Имя же задает «пространство имен», в которое будут сгруппированы сессии, запущенные разными пользователями. Один клиент может иметь сразу несколько активных пространств имен (то есть несколько имен групп сессий).

- `string session_id([string $sid])`

Функция возвращает текущий идентификатор сессии `SID`. Если задан параметр `$sid`, то у активной сессии изменяется идентификатор

на `$sid`. Делать это не рекомендуется. Фактически, вызвав `session_id()` до `session_start()`, можем подключиться к любой (в том числе и «чужой») сессии на сервере, если знаем ее идентификатор.

Другие функции.

- `bool session_is_registered(string $name)`

Функция `session_is_registered()` возвращает значение `true`, если переменная с именем `$name` была зарегистрирована в сессии, иначе возвращается `false`.

- `bool session_unregister(struing $name)`

Функция отменяет регистрацию для переменной с именем `$name` для текущей сессии. При завершении сценария все будет выглядеть так, словно переменная с именем `$name` и не была никогда зарегистрирована. Возвращает `true`, если все прошло успешно, и `false` – в противном случае. После вызова функции `session_unregister()` глобальная переменная, которая была «аннулирована», не уничтожается, а сохраняет свое значение.

- `void session_unset()`

Функция `session_unset()`, в отличие от `session_unregister()`, не только отменяет регистрацию переменных (*всех* переменных сессии, а не какой-то одной), но и уничтожает глобальные переменные, которые были зарегистрированы в сессии.

- `string session_save_path([string $path])`

Функция возвращает имя каталога, в котором будут помещаться файлы – временные хранилища данных сессии. В случае, если указан параметр, активное имя каталога будет переустановлено на `$path`. При этом функция вернет предыдущий каталог.

Установка обработчиков сессии.

Если стандартные обработчики сессии не устраивают, например, нужно хранить переменные сессии в базе данных или еще где-то. В этом случае достаточно будет переопределить обработчики своими собственными функциями.

Обзор обработчиков. Всего существует 6 функций, связанных с сессиями, которые PHP вызывает в тот или иной момент работы механизма обработки сессий.

- `bool handler_open(string $save_path, string $session_name)`

Функция вызывается, когда вызывается `session_start()`. Обработчик должен взять на себя всю работу, связанную с открытием базы данных для группы сессий с именем `$session_name`. В параметре `$save_path` передается то, что было указано при вызове

session_save_path() или же путь к файлам-хранилищам данных сессий по умолчанию.

- bool handler_close()

Этот обработчик вызывается, когда данные сессии уже записаны во временное хранилище и его нужно закрыть.

- string handler_read(string $sid)

Вызов обработчика происходит, когда нужно прочитать данные сессии с идентификатором $sid из временного хранилища. Функция должна возвращать данные сессии в специальном формате, который выглядит так:

имя1=значение1; имя2=значение2; ...;

Здесь имяN задает имя очередной переменной, зарегистрированной в сессии, а значениеN – результат вызова функции Serialize() для этой переменной. *Пример*: foo|i:1;count|i:10;

Она говорит о том, что из временного хранилища были прочитаны две целые переменные, первая из которых равна 1, а вторая – 10.

- string handler_write(string $sid, string $data)

Обработчик предназначен для записи данных сессии с идентификатором $sid во временное хранилище – например, открытое ранее обработчиком handler_open(). Параметр $data задается задается аналогично описаному выше. Действия этой функции сводятся к записи в базу данных строки $data без каких-либо ее изменений.

- bool handler_destroy(string $sid)

Обработчик вызывается, когда сессия с идентификатором $sid должна быть уничтожена.

- bool handler_gc(int $maxlifetime)

Обработчик вызывается при завершении работы сценария. Если пользователь окончательно «покинул» сервер, значит, данные сессии во временном хранилище можно уничтожить. Ей передается в параметрах то время (в секундах), по прошествии которого PHP принимает решение о необходимости «собрать мусор» (garbage collection) – т. е., это максимальное время существования сессии. Если храним данные сессии в базе данных, должны удалить из нее все записи, доступ к которым не осуществлялся более, чем $maxlifetime секунд. Таким образом, «застарелые» временные хранилища будут иногда очищаться.

Регистрация обработчиков.

- void session_set_save_handler($open, $close, $read, $write, $destroy, $gc)

Функция регистрирует подпрограммы, имена которых переданы в ее параметрах, как обработчики текущей сессии. Параметр $open содержит имя функции, которая будет вызвана при инициализации сессии, а $close – функции, вызываемой при ее закрытии. В $read и $write нужно указать имена обработчиков для чтения и записи во временное хранилище. Функция с именем, заданным в $destroy, будет вызвана при уничтожении сессии. Обработчик, определяемый параметром $gc, используется как сборщик мусора. Функцию можно вызывать только *до* инициализации сессии, иначе она игнорируется.

Сессии и Cookies. До сих пор подразумевалось, что использование сессий немыслимо без Cookies. Действительно, Cookies представляют собой наиболее простое решение задачи идентификации каждого подключившегося пользователя, что необходимо для связи временного хранилища и данных сессии. Но как быть, если пользователи отключили Cookies в своих браузерах? На любом браузере нужен механизм, позволяющий отказаться от использования Cookies при управлении сессиями. Такой механизм существует в PHP, и основная его идея состоит в том, чтобы передавать идентификатор сессии не в Cookies, а например, в данных запроса GET.

Явное использование константы SID. В PHP существует специальная константа с именем SID. Она содержит имя группы текущей сессии и ее идентификатор в формате имя=идентификатор. Достаточно передать значение константы SID в сценарий, чтобы он «подумал», что данные пришли из Cookies.

Пример использования сессий без Cookies

```
<?
session_name("test");
session_start();
session_register("count");
$count=@$count+1;
?>
<body>
<h2>Счетчик</h2>
В текущей сессии работы с браузером Вы открыли эту страницу <?=$count?> раз(а). Закройте браузер, чтобы обнулить этот счетчик.<hr>
<a href=sesget.php?<?=SID?>>Click here!</a>
</body>
```

Если набрать в браузере адрес:

http://www.somehost.ru/sesget.php

то создастся новая сессия с уникальным идентификатором. Если сразу же нажать кнопку **Обновить**, счетчик не увеличится, потому что при каждом запуске будет создаваться новое временное хранилище – у PHP просто нет информации об идентификаторе пользователя. В предпоследней строчке передаются в сценарий, запускаемый через гипер-

ссылку, данные об идентификаторе текущей сессии. Теперь с его точки зрения они якобы пришли из Cookies…

Все будет работать так, как описано, только в том случае, если в браузере действительно отключены Cookies. Если же они включены, PHP просто не будет генерировать константу `SID` (она будет пустой) и задействует Cookies.

Неявное изменение гиперссылок. Если в гипессылке по ошибке пропустить `<?=SID?>`, PHP вставит его автоматически. Причем так, чтобы это не повредило другим параметрам, уже присутствующим в URL. Если запустить следующий сценарий в браузере, навести мышь на гиперссылку и посмотреть строку состояния,

```
<?session_start()?>
<body>
<a href=/path/some.php>Click here!</a><br>
<a href=/path/some.html?a=a&b=b>Click here!</a><br>
<a href=/>Click here!</a><br>
</body>
```

то получим следующие адреса этих ссылок с точки зрения браузера:

```
http://www.some.ru/path/some.php?PHPSESSID=816a921f
http://www.some.ru/path/some.html?a=a&b=b&PHPSESSID
=86a20
http://www.some.ru/?PHPSESSID=8114536a920bfb2a
```

Во втором адресе идентификатор корректно вставился в конец обычных параметров страницы. В третьем адресе идентификатор сессии прикрепляется к URL независимо от типа документа, на который он указывает. Таким образом, константа `SID` – это устаревший прием передачи идентификатора сессии.

Неявное изменение формы. PHP умеет не только изменять гиперссылки, он также и добавляет скрытые поля в формы, которые формирует сценарий, чтобы передать идентификатор сессии вызываемому документу! Пример сценария, который выводит пустую форму, и в ней появляется дополнительное скрытое поле с идентификатором сессии.

```
<?session_start()?>
<form action=aaa method=post>
</form>
```

То, что выдается в браузере (Internet Explorer) после запуска этого сценария и выбора в меню пункта *Просмотр в виде HTML:*

```
<form action="aaa" method="post">
<INPUT TYPE=HIDDEN NAME="PHPSESSID"
VALUE="0a717e848e91db11b524a">
</form>
```

PHP добавил в форму скрытое поле с нужным именем и значением. Он также заключил в кавычки значения атрибутов тэга `<form>`.

Примечание. Сценарию, рассчитанному на сессии все равно, включены Cookies в браузере пользователя, или нет. PHP автоматически добавляет идентификатор сессии ко всем ссылкам и формам, которые он встретит, сценарии все равно будут продолжать работать, даже если Cookies будет отключены, только их URL (да и всех других документов) немного удлинятся.

Тема 14 Работа с базой данных MYSQL

14.1 Соединение с базой данных
14.2 Выполнение запросов к базе данных

14.1 Соединение с базой данных

База данных – совокупность связанных данных, сохраняемая в двумерных таблицах информационной системы. Программное обеспечение информационной системы, обеспечивающей создание, ведение и совместное использование баз данных, называется системой управления базами данных (СУБД).

С точки зрения программы база данных MySQL представляет собой организованный набор поименованных *таблиц*. Каждая таблица состоит из *записей*. Запись может содержать одно или несколько именованных *полей*. Число и имена полей задаются при создании таблицы. Каждое поле имеет определенный *тип.*

Преимущества работы с базами данных по сравнению с файлами:

1 Легко сортировать записи по дате/времени, организовывать поиск, различные отборы записей.

2 Нет проблем с совместным доступом к данным.

3 Работа с базами данных происходит быстрее, чем с файлами.

Устройство MySQL. MySQL – одна из самых популярных СУБД, которые используются в Web-программированиию Она предназначена для создания небольших (не более 100 Мбайт) баз данных, и поддерживает некоторое подмножество языка запросов SQL. *SQL* – специально разработанный стандарт языка запросов к базам данных.

MySQL – это программа-сервер, постоянно работающая на компьютере. Клиентские программы посылают ей специальные *запросы* через механизм сокетов (то есть при помощи сетевых средств), она их обрабатывает и запоминает результат. Затем, также по специальному запросу клиента, весь этот результат или его часть передается обратно.

Механизм использования сокетов подразумевает технологию *клиент-сервер*, а это означает, что в системе должна быть запущена специальная программа – MySQL-сервер, которая принимает и обрабатывает запросы от программ.

Структура MySQL трехуровневая: базы данных – таблицы – записи. Один сервер MySQL может поддерживать сразу несколько баз данных, доступ к которым может разграничиваться логином и паролем.

Соединение с базой данных. Прежде чем работать с базой данных, необходимо установить с ней сетевое соединение, а также провести авторизацию пользователя. Для этого служит функция `mysql_connect()`.

- `int mysql_connect([string $hostname] [,string $username][,string $password])`

Функция `mysql_connect()` устанавливает сетевое соединение с базой данных MySQL, расположенной на хосте `$hostname` (по умолчанию это `localhost`, т. е. текущий компьютер), и возвращает идентификатор открытого соединения. Вся дальнейшая работа ведется с этим идентификатором. При регистрации указывается имя пользователя `$username` и пароль `$password` (по умолчанию имя пользователя, от которого запущен текущий процесс, и пустой пароль). Строка `$hostname` также может включать в себя номер порта в формате: `имя_хоста:порт` (если сервер MySQL настроен не на стандартный, а на какой-то другой порт). При следующем запуске функции с теми же самыми аргументами второе соединение не будет открыто, а функция возвратит идентификатор уже существующего.

Соединение с MySQL-сервером будет автоматически закрыто по завершении работы сценария, либо при вызове функции `mysql_close()`.

Если планируется открывать только одно соединение с базой данных за все время работы сценария, то можно не сохранять возвращенное значение, а также не указывать идентификатор соединения при вызове всех остальных функций.

До того как послать первый запрос серверу MySQL, необходимо указать, с какой базой данных необходимо работать. Для этого предназначена функция `mysql_select_db()`.

- `int mysql_select_db(string $dbname [,int $link_identifier])`

Она уведомляет PHP, что в дальнейших операциях с соединением `$link_identifier` (или с последним открытым соединением, если указанный параметр не задан) будет использоваться база данных `$dbname`.

Обработка ошибок. Если в процессе работы с MySQL возникают ошибки, то сообщение об ошибке и ее номер можно получить с помощью следующих двух функций.

- `int mysql_errno([int $link_identifier])`

Функция возвращает номер последней зарегистрированной ошибки. Идентификатор соединения `$link_identifier` можно не указывать, если за время работы сценария было установлено только одно соединение.

- `string mysql_error([int $link_identifier])`

Эта функция возвращает не номер, а строку, содержащую текст сообщения об ошибке. Ее удобно применять в отладочных целях.

Выполнение запросов к базе данных. Для формирования запросов к базе данных существует функция – `mysql_query()`, которая возвращает идентификатор результирующего набора данных.

- `int mysql_query (string $query [,int $link_identifier])`

Эта функция посылает MySQL-серверу запрос `$query` и возвращает идентификатор ответа, или результата. Параметр `$query` представляет собой строку, составленную по правилам языка SQL. Используется установленное ранее соединение `$link_identifier`, а в случае его отсутствия – последнее открытое соединение.

Есть несколько команд SQL, которые возвращают только признак, успешно они выполнились или нет (это команды `UPDATE`, `INSERT` и т. д.). В этом случае этот признак и будет возвращен функцией. Для запроса `SELECT` возвращается идентификатор вывода, нулевое значение которого свидетельствует о том, что произошла ошибка.

Существует еще одна функция для выполнения запроса, но использовать ее менее удобно, поскольку всякий раз приходится указывать имя базы данных, к которой осуществляется доступ.

- `int mysql(string $dbname, string $query [,int $link_identifier])`

Служит для тех же целей, что и функция mysql_query(), только обращение осуществляется не к текущей выбранной базе данных, а к указанной в параметре `$dbname`. Параметр `$link_identifier` можно опустить, тогда используется последнее открытое соединение.

14.2 Выполнение запросов к базе данных

Все запросы к базе данных посылаются при помощи функции – `mysql_query()` или `mysql()`. Они должны передаваться ей в виде строкового параметра. Этот параметр может быть и многострочным – т. е., содержать символы перевода строки. MySQL допускает включение любого количества пробелов, символов табуляции или перевода строки везде, где разрешено использование одного пробела.

Создание таблицы

- `create table ИмяТаблицы (ИмяПоля тип, ИмяПоля тип, ...)`

В базе данных создается новая таблица с колонками (полями), определяемыми своими именами (`ИмяПоля`) и указанными *типами*.

Типы полей. В квадратные скобки заключены необязательные элементы.

Целые числа. Формат записи:
`префиксINT [UNSIGNED]`

Необязательный флаг `UNSIGNED` задает, что будет создано поле для хранения беззнаковых чисел (больших или равных 0). Имена типов:

`TINYINT` – может хранить числа от –128 до +127
`SMALLINT` – диапазон от –32 768 до 32 767
`MEDIUMINT` – диапазон от –8 388 608 до 8 388 607
`INT` – диапазон от –2 147 483 648 до 2 147 483 647
`BIGINT` – диапазон от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

Дробные числа. Формат записи:
`ИмяТипа[(length,decimals)] [UNSIGNED]`

Здесь `length` – ширина поля, `decimals` – количество знаков после десятичной точки, которые будут учитываться. `UNSIGNED` задает беззнаковые числа.

`FLOAT` – число с плавающей точкой небольшой точности
`DOUBLE` – число с плавающей точкой двойной точности
`REAL` – синоним для DOUBLE
`DECIMAL` – дробное число, хранящееся в виде строки
`NUMERIC` – синоним для DECIMAL

Строковые типы данных. Строки представляют собой массивы символов. Формат записи:
`VARCHAR(length) [BINARY]`

Тип строки, которая может хранить не более `length` символов, где `length` принадлежит диапазону от 1 до 255. При занесении некоторого значения в поле такого типа из него автоматически вырезаются концевые пробелы. Если указан флаг `BINARY`, то при запросе `SELECT` строка будет сравниваться с учетом регистра. Тип `VARCHAR` неудобен тем, что может хранить не более 255 символов.

`TINYTEXT` – может хранить максимум 255 символов
`TEXT` – может хранить не более 65 535 символов
`MEDIUMTEXT` – может хранить максимум 16 777 215 символов
`LONGTEXT` – может хранить 4 294 967 295 символов

Бинарные данные. Бинарные данные – это почти то же самое, что и данные в формате `TEXT`, но только при поиске в них учитывается регистр символов. Имеется 4 типа бинарных данных:

`TINYBLOB` – может хранить максимум 255 символов
`BLOB` – может хранить не более 65 535 символов
`MEDIUMBLOB` – может хранить максимум 16 777 215 символов
`LONGBLOB` – может хранить 4 294 967 295 символов

BLOB-данные не перекодируются автоматически, если при работе с установленным соединением включена возможность перекодирования текста «на лету».

Дата и время.

DATE – дата в формате ГГГГ-ММ-ДД

TIME – время в формате ЧЧ:ММ:СС

DATETIME – дата и время в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС

TIMESTAMP – время и дата в формате timestamp. Оно отображается в виде ГГГГММДДЧЧММСС.

Перечисления и множества. Тип перечисления задает, что значение соответствующего поля может быть не любой строкой или числом, а только одним из нескольких указанных при создании таблицы значений: value1, value2 и т. д. Формат задания перечисления:

```
ENUM(value1,value2,value3,...)
```

Множества означают, что в соответствующем поле может содержаться не одно, а сразу несколько значений (value1, value2 и т.д., т. е. – множество значений). Формат задания перечисления:

```
SET(value1,value2,value3,...)
```

Значений во множестве может быть не более 64 штук.

Модификаторы и флаги типов. К типу можно также присоединять модификаторы, которые задают его «поведение» и те операции, которые можно (или, наоборот, запрещено) выполнять с соответствующими столбцами. *Основные модификаторы MySQL:*

not null – означает, что поле не может содержать неопределенное значение – в частности, поле обязательно должно быть инициализировано при вставке новой записи в таблицу (если не задано значение по умолчанию)

primary key – отражает, что поле является первичным ключом, т. е. идентификатором записи, на которой можно ссылаться auto_increment. При вставке новой записи поле получит уникальное значение, так что в таблице никогда не будут существовать два поля с одинаковыми номерами.

Default – задает значение по умолчанию для поля, которое будет использовано, если при вставке записи поле не было проинициализировано явно.

Удаление таблицы:

- drop table ИмяТаблицы

Удаляет таблицу ИмяТаблицы. Таблица не обязательно должна быть пустой.

Вставка записи:

- insert into ИмяТаблицы(ИмяПоля1 ИмяПоля2 ...) values('зн1','зн2',...)

Добавляет в таблицу ИмяТаблицы запись, у которой поля, обозначенные как ИмяПоляN, установлены в значения, соответственно, знN. Те поля, которые в этой команде не перечислены, получают «неопределенные» значения (неопределенное значение – это не пустая строка, а просто признак, который говорит MySQL, что у данного поля нет никакого значения). Если для не указанного здесь поля при создании таблицы был задан not null, то данная команда закончится неуспешно. Значения полей можно заключать в апострофы и в обычные кавычки. При вставке в таблицу бинарных данных (или текстовых, содержащих апострофы и слэши) некоторые символы \, ' и символ с нулевым кодом должны быть «защищены» обратными слэшами.

Удаление записей:

- delete from ИмяТаблицы where Выражение

Удаляет из таблицы ИмяТаблицы все записи, для которых выполнено Выражение. Параметр Выражение – это логическое выражение, например

```
(id<10) and (name regexp 'a*b') and (age=25).
```

Поиск записей:

- select * from Таблица where Выражение [order by ИмяПоля [desc]]

Команда предназначена для того, чтобы искать все записи, удовлетворяющие выражению Выражение. Если записей несколько, то при указанном предложении order by они будут отсортированы по тому полю, имя которого записывается правее этого ключевого слова (если задан описатель desc, то упорядочивание происходит в обратном порядке). В предложении order by могут задаваться несколько полей. Символ * предписывает, что из отобранных записей следует извлечь *все* поля, когда будет выполнена команда получения выборки. Вместо звездочки можно через запятую непосредственно перечислить имена полей, которые требуют извлечения.

Обновление записей:

- update Таблица set (ИмяПоля1='зн1', ИмяПоля1='зн2', ...) where Выражение

В таблице Таблица для всех записей, удовлетворяющих выражению Выражение, указанные поля устанавливаются в соответствующие значения.

Получение числа записей, удовлетворяющих выражению:

```
select count(if(Выражение,1,NULL)) from Таблица
```

Получение уникальных значений столбцов. При использовании базы данных часто бывает нужным узнать, какие *уникальные* значения существуют в данном столбце таблицы. Например, если у каждой за-

писи в некоей статистической таблице, содержащей сведения о людях, у нас есть поле Country (страна), в котором указана страна проживания конкретного человека, и мы хотим выяснить, в каких же странах проживают все люди, дожившие до 30 лет, занесенные в таблицу, можно выполнить запрос:

```
select distinct Country from Таблица where age>=30
```

Этот запрос сгенерирует результат, состоящий из одного столбца, в котором и будут перечислены искомые страны.

Получение результата. Результатом выполнения запроса является *набор данных* и количество вошедших в него записей. Каждая запись – это *список значений полей* в том же порядке, в котором они были указаны в запросе `select ... from Таблица` на месте многоточия (если там была звездочка, то все поля). Таким образом, результат – это своеобразный двумерный массив: первый индекс – номер записи и второй – имя поля.

Параметры результата:

- `int mysql_num_rows(int $result)`
  Функция возвращает число записей в результате запроса.
- `int mysql_num_fields(int $result)`
  Функция возвращает число полей в одной строке результата.

Получение поля результата:

- `int mysql_result (int $result, int $row, mixed $field)`

Функция возвращает значение поля `$field` в строке результата с номером `$row`. Параметр `$field` может задавать не только имя поля, но и его номер – позицию, на которой столбец «стоял» при создании таблицы.

Получение целой строки результата. Этот способ получения результата чем-то похож на работу с файлами: появляется понятие текущей записи результата, и следующая операция считывания передвигает этот указатель на одну позицию вперед.

- `array mysql_fetch_row(int $result)`

Функция возвращает массив-список со значениями полей очередной строки результата `$result`. Если указатель текущей позиции результата был установлен за последней записью, возвращает `false`. Текущая позиция сдвигается к следующей записи, так что очередной вызов `mysql_fetch_row()` вернет следующую строку результата.
Пример:

```
$r=mysql_query ("select * from OurTable where age<30");
while($Row=mysql_fetch_row($r))
{ // обрабатываем строку $Row }
```


Цикл оборвется, как только строки закончатся, т. е. когда `mysql_fetch_row()` вернет `false`.

Работать с числовыми индексами полей не удобно. Лучше использовать для адресации поля внутри результата его имя.

- `array mysql_fetch_array(int $result)`

Функция возвращает очередную строку результата в виде ассоциативного массива, где каждому полю сопоставлен элемент с ключом, совпадающим с именем поля. Дополнительно в массив записываются элементы с числовыми ключами и значениями, соответствующими величинам полей с этими индексами. В возвращаемом массиве они размещаются сразу за элементами с «обычными» ключами.

- `int mysql_data_seek (int $result, int $row_number)`

Функция устанавливает указатель текущей строки в результате `$result` в позицию `$row_number`, так что следующий вызов `mysql_fetch_row()` и `mysql_fetch_array()` вернет значения полей именно этой строки. Возвращает `false` в случае ошибки или если строки кончились.

Получение информации о результате:

- `string mysql_field_name(int $result, int $field_index)`

Функция возвращает *имя* поля, которое расположено в результате по смещению `$field_index`.

- `string mysql_field_type(int $result, int $field_offset)`

Функция возвращает *тип* соответствующей колонки в результате. Им может быть, например, `int`, `double` и т. д.

- `int mysql_field_len(int $result, int $field_offset)`

Функция возвращает *длину* поля в результате `$result`. Поле задается указанием его смещения. Под длиной подразумевается тот размер, который был указан при его создании. Например, если полеимеет тип `varchar` и было создано (вместе с таблицей) с типом `varchar(100)`, то для него будет возвращено 100.

- `string mysql_field_flags(int $result, int $field_offset)`

Функция возвращает флаги, которые были использованы при создании указанного поля в таблице. Возвращаемая строка представляет собой набор слов, разделенных пробелами, которую можно преобразовать в массив при помощи функции `explode()`:

```
$Flags=explode(" ", mysql_field_flags($r,
```

```
$field_offset));
```

Флаги типов полей:

`not_null` – поле обязательно должно быть проинициализировано при вставке очередной строки в таблицу.

`Primary_key` – поле является первичным ключом – т. е. идентификатором строки, который будет использован для ссылок на нее.

`Unique_key` – поле должно быть уникальным.

`Multiple_key` – по этому полю построен индекс.

`Blob` – поле может содержать бинарный блок данных, который никак не интерпретируется.

`Unsigned` – поле содержит беззнаковые числа.

`zerofill` – использовать символы с нулевым кодом вместо пробелов.

`binary` – поле содержит бинарные данные.

`enum` – поле содержит элемент перечисления, т. е. только один элемент из нескольких возможных.

`auto_increment` – это поле автоматически нумеруется. Проставляется MySQL при добавлении новой записи так, чтобы в таблице никогда не образовывалось нескольких строк с одинаковым значением этого поля.

`timestamp` – в поле динамически проставляется текущее время при добавлении или изменении записи.

- `string mysql_field_table (int $result, int $field_offset)`

Возвращает имя таблицы, из которой было извлечено поле со смещением `$field_offset` в результате `$result`. Результат запроса может быть получен из нескольких таблиц.

Пример использования функций поддержки MySQL:

```
<?
// Соединяемся с сервером на локальной машине
mysql_connect("localhost");
// Выбираем текущую базу данных
mysql_select_db("my_database");
// Получаем все данные таблицы
$result = mysql_query("SELECT * FROM tbl");
// Запрашиваем идентификатор данных о полях таблицы
$fields = mysql_num_fields($result);
// Узнаем число записей в таблице
$rows = mysql_num_rows($result);
// Получаем имя таблицы (правда, мы его и так знаем, но все же...)
$table = mysql_field_table($result,0);
```



```
echo "Таблица '$table' содержит $fields колонок и
$rows строк<BR>";
echo "Таблица содержит следующие поля:<BR>";
// "Проходимся" по всем полям и выводим информацию о них
for($i=0; $i<$fields; $i++) {
$type = mysql_field_type($result, $i);
$name = mysql_field_name($result, $i);
$len = mysql_field_len($result, $i);
$flags = mysql_field_flags($result, $i);
echo "$type $name $len $flags<BR>\n";
}
?>
```

Уникальные идентификаторы в MySQL. Обычно в таблице содержится много записей с разными значениями полей. Встает проблема выбора одной конкретной записи из этого массива. В рассмотренном примере таблицы с информацией о гражданах, запись можно однозначно идентифицировать по фамилии человека. Ну а если встретятся однофамильцы? Тогда по имени. А если же и имена совпадут? Во избежание недоразумений подобного рода в таблицу вводят еще одно вспомогательное поле (колонку), назвав ее, скажем, `id`. Этот `id` уникален у каждой записи, поэтому можно, зная `id` нужного человека, тут же получить его данные. Кроме того, если понадобится, например, зафиксировать в таблице еще и родственные связи людей (кто является чьим отцом, например), мы можно завести в ней еще одно поле – `parent_id`, в котором будет храниться `id` родителя.

Пусть необходимо добавить в таблицу сведения о еще одном человеке. Логично было бы, чтобы его `id` проставлялся автоматически. Для этого в MySQL предназначена возможность под названием `AUTO_INCREMENT`. При создании таблицы можно какое-нибудь ее поле (в нашем случае `id`) объявить так:

```
ИмяПоля int auto_increment primary key
```

Теперь любая операция `INSERT` автоматически проставит у добавленной записи поле `ИмяПоля` так, чтобы оно было уникально во всей таблице – MySQL это гарантирует. В простейшем случае – просто увеличит на 1 некий внутренний счетчик, глобальный для всей таблицы, и занесет его новое значение в нужное поле записи. Причем гарантируется, что никакие проблемы с совместным доступом к таблице просто не могут возникнуть.

Получить только что вставленный идентификатор можно при помощи функции `mysql_insert_id()`.

- `int mysql_insert_id([int $link_identifier])`

Функция возвращает непосредственно перед ее вызовом сгенерированный идентификатор записи для автоинкрементного поля после выполнения команды `insert`. *Пример*:

```
mysql_query("insert into Таблица (поле1, поле2)
values('aa','bb')");
$id=mysql_insert_id();
```

Теперь к только что вставленной записи можно обратиться, используя идентификатор `$id`:

```
$r=mysql_query("select * from Таблица where
id=$id");
$Row=mysql_fetch_array($r);
```

Функции работа с таблицами в целом:

- `int mysql_list_fields(string $dbname, string $tblname [,int $link])`

Функция возвращает информацию об указанной таблице `$tblname` в базе данных `$dbname`, используя идентификатор соединения `$link`, если он задан (в противном случае – последнее открытое соединение). Возвращаемое значение – идентификатор результата, который может быть проанализирован обычными средствами, либо при помощи функций `mysql_field_flags()`, `mysql_field_len()`, `mysql_field_name()` и `mysql_field_type()`. В случае ошибки возвращается `-1`.

- `int mysql_list_tables(string $database [,int $link_identifier])`

Функция возвращает идентификатор результата (одна колонка), в котором содержатся имена всех таблиц, присутствующих в базе данных. Для извлечения этих имен можно использовать функцию `mysql_result()` с номером колонки, равным 0.

Тема 15 Загрузка файлов на сервер

15.1 Multipart-формы, тег выбора файла
15.2 Поддержка закачки в PHP

15.1 Multipart-формы, тег выбора файла

Слово «закачать» обозначает загрузку файла клиента *на сервер*, и термин «скачать» – обратный процесс (*с сервера – клиенту*).

В большинстве случаев данные из формы в браузере, передающиеся методом `GET` или `POST`, приходят в формате:

```
поле1=значение1&поле2=значение2&...
```

При этом все символы, отличные от «английских» букв и цифр (и еще некоторых) URL-кодируются: заменяются на `%XX`, где `XX` – шестнадцатеричный код символа. Это замедляет закачку больших файлов. Multipart-формы используются для решения этой проблемы. Нужно в соответствующем тэге `<form>` задать параметр:

```
enctype=multipart/form-data
```

После этого данные, полученные от формы, будут разбиты на несколько блоков информации (по одному на каждый элемент формы). Каждый такой блок очень похож на обычную посылку «заголовки-данные» протокола HTTP:

```
-----------------Идентификатор_начала\n
Content-Disposition: form-data; name="имя" [;другие
параметры]\n
\n
значение\n
```

Браузер автоматически формирует строку `Идентификатор_начала` из расчета, чтобы она не встречалась ни в одном из передаваемых файлов (и ни в одном из других полей формы). Это означает, что сегодня идентификатор будет одним, а завтра – другим.

Для того, чтобы в форме появился элемент управления загрузкой файла – текстовое поле с кнопкой *Browse* справа, нужно в форму добавить следующий тег:

```
<input type=file name=имя_элемента
                  [size=размер_поля]>
```

Сценарию вместе с содержимым файла передается и некоторая другая информация, а именно: размер файла; имя файла в системе клиента; тип файла.

15.2 Поддержка закачки в PHP

Простые имена полей закачки. Интерпретатору не имеет значения, в каком формате приходят данные из формы. Он умеет их обраба-

тывать и «рассовывать» по переменным в любом формате. Однако данные одного специального поля формы – поля закачки – он интерпретирует особым образом. Пусть есть multipart-форма, а в ней – поле закачки файла:

```
<form action="script.php" method=POST
enctype=multipart/form-data>
<input type=file name="MyFile">
<input type=submit>
</form>
```

После выбора в этом поле нужного файла и отправки формы (и загрузки на сервер того файла, который был указан) PHP определит, что нужно принять файл, и сохранит его во временном каталоге на сервере. Кроме того, в программе создадутся несколько переменных:

`$MyFile` – имя временного файла на машине сервера, который содержит данные, переданные пользователем. Теперь этот файл можно удалять, копировать, переименовывать.

`$MyFile_name` – исходное имя файла, которое он имел до отправки на сервер.

`$MyFile_size` – размер закаченного файла в байтах.

`$MyFile_type` – тип загруженного файла, если браузер смог его определить. К примеру, `image/gif`, `text/html` и т. д.

Префикс у всех созданных переменных один и тот же – `MyFile_`. Этот префикс состоит из имени элемента закачки в форме, к которому присоединен знак `_`. Теперь можно, например, скопировать полученный файл при помощи `Copy($MyFile,"uploaded.dat")`.

Если процесс окончится неуспешно, можно определить это по отсутствию файла, имя которого задано в `$MyFile`, или же по отсутствию самой этой переменной в программе.

Пример сценария, представляющий собой простейший фотоальбом с возможностью добавления в него новых фотографий.

```
<?
$ImgDir="img"; // Каталог для хранения изображений
@mkdir($ImgDir,666); // Создаем, если его еще нет
// Проверяем, нажата ли кнопка добавления фотографии
if(@$doUpload) {
// Проверяем, принят ли файл
if(file_exists($File)) {
// Все в порядке – добавляем файл в каталог с фотографиями
// Используем то же имя, что и в системе пользователя
Copy($File,"$ImgDir/".basename($File_name));
}
}
```



```
// Теперь считываем в массив фотоальбом
$d=opendir($ImgDir); // открываем каталог
$Photos=array(); // изначально альбом пуст
// Перебираем все файлы
while(($e=readdir($d))!==false) {
// Это изображение GIF, JPG или PNG?
if(!ereg("^(.*)\\.(gif|jpg|png)$",$e,$P)) continue;
// Если нет, переходим к следующему файлу,
// иначе обрабатываем этот
$path="$ImgDir/$e"; // адрес
$sz=GetImageSize($path); // размер
$tm=filemtime($path); // время добавления
// Вставляем изображение в массив $Photos
$Photos[$tm] = array(
'time' => filemtime($path), // время добавления
'name' => $e, // имя файла
'url' => $path, // его URI
'w' => $sz[0], // ширина картинки
'h' => $sz[1], // ее высота
'wh' => $sz[3] // "width=xxx height=yyy"
);
}
// Ключи массива $Photos – время в секундах, когда была добавлена
// та или иная фотография. Сортируем массив: наиболее «свежие»
// фотографии располагаем ближе к его началу.
krsort($Photos);
// Данные для вывода готовы.
?>
<body>
<form action=photo.php method=POST
enctype=multipart/form-data>
<input type=file name=File><br>
<input type=submit name=doUpload value="Закачать
новую фотографию">
</form>
<?foreach($Photos as $n=>$Img) {?>
<img src=<?=$Img['url']?>
<?=$Img['wh']?>
alt="Добавлена
<?=date("d.m.Y H:i:s",$Img['time'])?>">
<?}?>
</body>
```

Сложные имена полей. Элементы формы могут иметь имена, выглядящие, как элементы массива: `A[10]`, `B[1][text]` и т. д. Рассмотрим, какие переменные создаст PHP при ее отправке на сервер при использовании сложных имен полей для отправки файла:

```
<form action="script.php" method=POST
enctype=multipart/form-data>
<h3>Выберите тип файлов в вашей системе:</h3>
Текстовый файл:
<input type=file name="File[text]"><br>
Бинарный файл:
<input type=file name="File[bin]"><br>
Картинка: <input type=file name="File[pic]"><br>
<input type=submit name=Go value="Отправить файлы">
</form>
```

После того как программа `script.php` примет данные из формы, PHP создаст для нее следующие переменные: ассоциативный массив `$File`, ключи которого – `text`, `bin` и `pic`, а соответствующие значения – имена временных файлов на сервере, созданных PHP при загрузке; массив `$File_name` с теми же ключами и значениями – именами файлов в системе пользователя; массив `$File_type` с теми же ключами и значениями – типами соответствующих файлов; массив `$File_size` со значениями – размерами этих файлов.

Информация об индексах в именах полей формы попала в ключи соответствующих массивов и сохранилась в них.

Описанный механизм работает замечательно, лишь когда задействуем элементы *одномерных* массивов в качестве имен полей формы. В случае же многомерных массивов возникают проблемы. *Пример*:

```
<form action="script.php" method=POST
enctype=multipart/form-data>
<input type=file name="File[a][b]">
<input type=submit>
</form>
```

При приеме данных такой формы PHP «запутается» и, хотя и создаст массив `$File`, но не поместит в него никаких полезных данных. В элемент с ключом `a` вместо имени файла попадает какое-то комплексное число. Однако PHP, помимо установки вышеперечисленных переменных, создает также глобальный массив с именем `$HTTP_POST_FILES`. В этом массиве содержатся верные данные, какое бы имя не имело поле закачки в форме.

Массив `$HTTP_POST_FILES` создается не всегда, а только в том случае, если в настройках PHP задействован параметр `track_vars`.

ЛИТЕРАТУРА

1 PHP и MySQL. Подсказки. Советы. Приемы работы / М. Белянин. – М.: НТ Пресс, 2007.

2 Афонин, С. Программирование на языке PHP / С. Афонин. – М.: НТ Пресс, 2007.

3 Веллинг, Л. Разработка Web-приложений с помощью PHP и MySQL / Л. Веллинг, Л. Томсон. – М.: Вильямс, 2005.

4 Гилмор, В. PHP 4. Учебный курс / В. Гилмор. – СПб.: Питер, 2001.

5 Григин, И. PHP 4. Специальный справочник / И. Григин. – СПб.: Питер, 2002.

6 Дунаев, В. Самоучитель PHP / В. Дунаев. – СПб.: Питер, 2006.

7 Зандстра, М. Освой самостоятельно PHP4 за 24 часа / Мэт Зандстра. – М.: Вильямс, 2001.

8 Колисниченко, Д. Профессиональное программирование на PHP / Д. Колисниченко. – СПб.: БХВ-Петербург, 2007.

9 Косентино, Кр. PHP: Web-профессионалам / Кр. Косентино. – Киев: BHV-Киев, 2001.

10 Костарев, А. PHP в Web-дизайне / А Костарев. – СПб.: BHV-СПб, 2002.

11 Котеров, Дм. Самоучитель PHP4 / Дм. Котеров. – СПб.: BHV-СПб, 2001.

12 Кузнецов, М. Головоломки на PHP для хакера / М. Кузнецов, И. Симдянов. – СПб.: БХВ-Петербург, 2008

13 Кузнецов, С. PHP 4.0. Руководство пользователя / С. Кузнецов. – М.: Майор, 2001.

14 Куссуль, Н. Использование PHP. Самоучитель / Н. Куссуль, А. Шелестов. – М.: Диалектика, 2005.

15 Кухпрчик, А. PHP: обучение на примерах / А. Кухарчик. – М.: Новое знание, 2004.

16 Мазуркевич, А. PHP. Настольная книга программиста / А. Мазуркевич, Д. Еловой. – М.: Новое знание, 2006.

17 Мелони, Дж. PHP 4 в действии / Дж Мелони. – М.: Лучшие книги, 2002.

18 Ньюман, К. Освой самостоятельно PHP. 10 минут на урок / К. Ньюман. – М.: Вильямс, 2005.

19 Орлов, А.А. PHP. Полезные приемы / А.А. Орлов. – М.: Горячая Линия – Телеком, 2003.

20 Пол, Х. PHP. Справочник / Х. Пол. – М.: Кудиц-Образ, 2006.

21 Профессиональное PHP программирование / Джезус Кастаньетто [и др.]. – М.: Символ-Плюс, 2001.

22 Профессиональное PHP программирование / Крис Сколло [и др.]. – М.: Символ-Плюс, 2003.

23 Ратшиллер, Т. PHP4: разработка Web-приложений. Библиотека программиста / Т. Ратшиллер, Т. Геркен. – СПб.: Питер, 2003.

24 Савельева, Н. Основы программирования на PHP / Н. Савельева. – М.: ИНТУИТ.РУ, 2005.

25 Сидмянов, И.В. Головоломки на PHP для хакера / Симдянов И.В., Кузнецов М.В. – М.: BHV, 2006.

26 Скляр, Д. PHP. Рецепты программирования / Д. Скляр, А. Трахтенберг. – СПб.: BHV-Санкт-Петербург, 2007.

27 Томсон, Л. Разработка Web-приложений на PHP и MySQL / Л. Томсон, Л. Веллинг. – М.: ДиаСофтЮП, 2003.

28 Томсон, Л. Разработка Web-приложений на PHP и MySQL / Л. Томсон, Л. Веллинг. – М.: Диасофт, 2001.

29 Ульман, Л. Основы программирования на PHP. Самоучитель / Л. Ульман. – М.: ДМК Пресс, 2001.

30 Уэнц, К. PHP. Сборник готовых рецептов / К. Уэнц. – М.: Вильямс, 2006.

31 Фленов, М. PHP глазами хакера / М. Фленов. – М.: BHV, 2005.

32 Фролов, А. Практика применения Perl, PHP, Apache, MySQL для активных Web-сайтов / А. Фролов, Г. Фролов. – М.: Русская Редакция, 2002.

33 Харрис, Э. PHP/MySQL для начинающих / Э. Харрис. – М.: Кудиц-Образ, 2007.

34 Хольцнер, С. PHP в примерах / С. Хольцнер. – М.: Бином-Пресс, 2007

35 Черный, А.И. Самоучитель FLASH и PHP / А.И. Черный. – М.: Питер, 2004.

36 Швендимен, Бл. PHP 4. Руководство разработчика / Бл. Швендимен. – М.: Вильямс, 2002.

37 Шкрыль, А. PHP – это просто. Программируем для Web-сайта / А. Шкрыль. – М.: BHV, 2006.

38 Шлосснейгл, Дж. Профессиональное программирование на PHP / Дж. Шлосснейгл. – М.: Вильямс, 2005.

Учебное издание

Ружицкая Елена Адольфовна

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ПЛАТФОРМЕ NET: PHP.NET

Тексты лекций по спецкурсу для студентов специальности 1–40 01 01 «Программное обеспечение информационных технологий» специализации 1–40 01 01 01 «Компьютерные системы и Internet-технологии»

В авторской редакции

Подписано в печать 20.03.2008 г. (89). Формат издания 60x84 1/16. Бумага писчая №1. Печать на ризографе. Гарнитура Таймс. Усл.печ.л. 7,84. Уч-изд.л. 6,06. Тираж 20 экз.

Отпечатано в учреждении образования
«Гомельский государственный университет
имени Франциска Скорины»,
246019 г. Гомель, ул. Советская, 104