CS 85: PHP Programming

Santa Monica College Computer Science & Information Systems Dept.

Module 6: Files and Directories

Uploading Files

Understanding Directories

Write to File

Read to File

Open and Close files

- Retrieving the File Information

-Download Files

-Lock Files

Working with files is an important part of any programming language, and PHP is no different. Whatever your reasons are for wanting to manipulate files, PHP will happily accommodate them through the use of a handful of functions. Files are commonly used with PHP to store and retrieve data. Some examples would be user file uploads for processing, user submitted data stored in files, online data storage can use files to organized data.

# File Types

In PHP it is important to understand how files and directories are organized, secured and accessed. The vast majority of web server on the Internet are Linux systems. Therefore, web developer should have basic understand of the Linux command line and how the operating systems store, secure files and directories.

The Linux operating system has two main types of files that Web Developers generally interact with. The first is the binary files, it is a files created from a series characters that is non-human readable. The binary file system is used for storing images, sounds, executables programs or compressed data. The second files type is the text file. The text file storage human readable text characters with no special formatting such as bold or italic text. Linux, Windows and OSX each interpret text and binary files differently. A common issue user see with text files is the end of line character sequence that represents a new line. On windows the escape character \n\r represents a new line, on Linux a \n is a new line and on OSX \r is a new line. Why is this important for a back end web developer? When interpreting, outputting, or processing user submitted files, it is important to know and understand why different text files have different output cause by the new line escape characters’ issues.

Since the majority of web servers are running on Linux systems, this module will denote new lines with the \n escape characters. But when uploading Windows text files to Linux, be aware there might be an issues with the text formatting.

# Linux Permissions

Linux users exist to control the extent to which people and programs using the Linux system can control it. This is accomplished by a system of file permissions. Each file belongs to one of the users - that is, each file has an owner. Additionally, a file can be assigned to a group of users, but the owner must be a member of that group. Each file has three kinds of permissions: read, write and execute. These permissions can be assigned to three kinds of owner relations: owner, group and other. Other includes all users who are not the owner of the file and do not belong to the group which owns the file. Only the file owner or the Linux superuser (root) can change the permissions or ownership of a file.

This system allows precise control over who can do what on a given computer. Users can be prevented from modifying system files by removing the "write" permission from them, or from executing certain commands by removing the "execute" permission. Notice that users may be allowed to execute programs but not alter them.

File permissions are usually given as three octal digits (each from 0 to 7). The digits represent the permissions for, respectively, owner, group and other users. Each digit is the sum of permission codes: 1 for execute, 2 for write and 4 for read. For example, "755" allows everyone to read or execute the file, but only its owner can write it. "400" allows the owner to read the file, and no one else is allowed to do anything. "540" allows the owner to read or execute the file, group members to only read the file and other users to do nothing.

+-----+---+--------------------------+

| rwx | 7 | Read, write and execute |

| rw- | 6 | Read, write |

| r-x | 5 | Read, and execute |

| r-- | 4 | Read, |

| -wx | 3 | Write and execute |

| -w- | 2 | Write |

| --x | 1 | Execute |

| --- | 0 | no permissions |

+------------------------------------+

| Permission | Octal| Field |

+------------+------+-------+

| rwx------ | 700 | User |

| ---rwx--- | 070 | Group |

| ------rwx | 007 | Other |

+------------+------+-------+

+------------------------+-----------+

| chmod u=rwx,g=rwx,o=rx | chmod 775 | For world executables files

| chmod u=rwx,g=rx,o= | chmod 750 | For executables by group only

| chmod u=rw,g=r,o=r | chmod 644 | For world readable files

| chmod u=rw,g=r,o= | chmod 640 | For group readable files

| chmod u=rw,go= | chmod 600 | For private readable files

| chmod u=rwx,go= | chmod 700 | For private executables

+------------------------+-----------+

The built in Linux and PHP are the functions chmod (change mode), chown (change owner), and chgrp (change group). These functions are used to change file permissions, ownership and group the file or directories is associated with. The chmod() function is used to change the file or directory permissions of a file or directory the user running the command has permission to modify. The syntax of the chmod() function is chmod(filename, permission code). The filename is the name of the file the file permission will be change. The permission code is the permission level to change the file to. For example: 400 for only owner to has read permission, or 755 give owner full read, write execute and everyone else has read and execute. For this course leave the first leading number to zero, followed by the owner, group, and everyone permission.

chmod(“filename.txt”, 0755);

chmod(“webpage.php”, 0755);

# Permission Info

To gather information on files or directory permissions PHP has the built in function fileperms(). The fileperms() function returns the permissions for a file or directory. This function returns the permission as a number on success or FALSE on failure. The function takes a filename as the argument and return the permission in octal. To understand the permission, use the decot() function to convert the octal number to a decimal permission bitmap (ex 755, 400).

**<?php**$perminfo = *fileperms*($testfile);   
$perminfo = *decoct*($perminfo % 01000);   
**echo "file permissions for** $testfile**: 0"** . $perminfo . **"<br/>\n"**;  
**?>**

# Directories

PHP allows you to create, read, and move around a file system structure hosting your website. By using the built in PHP functions listed below you are able change to a specific directory, list the files in a directory, and make changes to the files in that directory.

A handle is a type of variable that PHP uses to represent a resource such as a file or directory.

|  |  |
| --- | --- |
| **Function** | **Description** |
| [chdir()](https://www.w3schools.com/php/func_directory_chdir.asp) | Changes the current directory |
| [chroot()](https://www.w3schools.com/php/func_directory_chroot.asp) | Changes the root directory |
| [closedir()](https://www.w3schools.com/php/func_directory_closedir.asp) | Closes a directory handle |
| [dir()](https://www.w3schools.com/php/func_directory_dir.asp) | Returns an instance of the Directory class |
| [getcwd()](https://www.w3schools.com/php/func_directory_getcwd.asp) | Returns the current working directory |
| [opendir()](https://www.w3schools.com/php/func_directory_opendir.asp) | Opens a directory handle |
| [readdir()](https://www.w3schools.com/php/func_directory_readdir.asp) | Returns an entry from a directory handle |
| [rewinddir()](https://www.w3schools.com/php/func_directory_rewinddir.asp) | Resets a directory handle |
| [scandir()](https://www.w3schools.com/php/func_directory_scandir.asp) | Returns an array of files and directories of a specified directory |

A directory pointer is a variable that points to a record in a directory listing. When a directory is initially opened with opendir(), the directory points is set to point to the start if the directory listing. The points move to the next entry in the directory with each iteration of readdir() call.

Sample Code

**<?php**$dir = **"/etc/php5/"**;  
  
*// Open a known directory, and proceed to read its contents***if** (*is\_dir*($dir)) {  
 **if** ($dh = *opendir*($dir)) {  
 **while** (($file = *readdir*($dh)) !== **false**) {  
 **echo "filename:** $file **: filetype: "** . *filetype*($dir . $file) . **"\n"**;  
 }  
 *closedir*($dh);  
 }  
}  
**?>**

# Scan Directory

Using the PHP scandir() function, developers are able to return back an index array of the names of the directories and files in the system path provided to the scandir() function call. The array would be sorted alphabetical order. If you only need a directory listing, this could replace the above code that uses the opendir(), closedir() and scandir() to gather the same information.

Sample Code

**<?php**$dir = **"/images/"**;  
  
*// Sort in ascending order - this is default*$a = *scandir*($dir);  
  
*// Sort in descending order*$b = *scandir*($dir,1);  
  
*print\_r*($a);  
*print\_r*($b);  
**?>**

# Create Directory

The built in PHP function mkdir() will create a directory in the file system path provided to the function. By passing only a directory name and not a full path, the mkdir() function will create a new folder in the current directory. The second required argument to the mkdir() function is the permissions to issue to the new directory. If the path provided points to a non-existing parent directory, an error message will be issued.

Syntax: mkdir(path,permissions)

**<?php***mkdir*(**"testing"**);  
*mkdir*(**"/etc/testing"**,0755);  
*mkdir*(**"/tmp/testing"**,0400);  
**?>**

To determine if a file or directory exist before attempting to write to it, use the function file\_exist(). The file\_exist() required a path to be passed to the function. The function will return a TRUE if the file or directory exist and FALSE if the file or directory does not exist.

Syntax: file\_exists(path)

**<?php  
echo** *file\_exists*(**"test.txt"**);  
**?>**

To check the permissions on files or directories use the below list of functions. The functions will let the developer know if the web server user running the PHP function on the host server have read, write, or execution permissions on the file or directory. Each function takes a path to a file or directory as the argument and return TRUE or FALSE.

Sample Code

**<?php**$file = **"setup.exe"**;  
**if**(*is\_executable*($file))  
{  
 **echo** (**"**$file **is executable"**);  
}  
**else**{  
 **echo** (**"**$file **is not executable"**);  
}  
**?>**

|  |  |
| --- | --- |
| [is\_dir()](https://www.w3schools.com/php/func_filesystem_is_dir.asp) | Checks whether a file is a directory |
| [is\_executable()](https://www.w3schools.com/php/func_filesystem_is_executable.asp) | Checks whether a file is executable |
| [is\_file()](https://www.w3schools.com/php/func_filesystem_is_file.asp) | Checks whether a file is a regular file |
| [is\_link()](https://www.w3schools.com/php/func_filesystem_is_link.asp) | Checks whether a file is a link |
| [is\_readable()](https://www.w3schools.com/php/func_filesystem_is_readable.asp) | Checks whether a file is readable |
| [is\_uploaded\_file()](https://www.w3schools.com/php/func_filesystem_is_uploaded_file.asp) | Checks whether a file was uploaded via HTTP POST |
| [is\_writable()](https://www.w3schools.com/php/func_filesystem_is_writable.asp) | Checks whether a file is writeable |
| [is\_writeable()](https://www.w3schools.com/php/func_filesystem_is_writeable.asp) | Alias of [is\_writable()](https://www.w3schools.com/php/func_filesystem_is_writable.asp) |

**<?php**$file = **"test.txt"**;  
**if**(*is\_readable*($file))  
{  
 **echo** (**"**$file **is readable"**);  
}  
**else**{  
 **echo** (**"**$file **is not readable"**);  
}  
**?>**

# Uploading to Web Server

It common for web pages to allow for uploading of images, file, or data for processing. Web sites such as Dropbox or Flicker allow users to upload images or files for storing. PHP allows for file uploads via HTML Forms. By using form POST method, the file content will be sent to the web server within the HTTP packet. The HTML form element attribute enctype="multipart/form-data" should be set. It specifies which content-type to use when submitting the form. The form input type should be set to file to allow for browsing and selecting a local system file. Once the form is submitted, the PHP action file will check file size, check if the file already exist, check the file format and if there is no errors will issue a call to the built in PHP function move\_uploaded\_file().

First create a HTML form an HTML form that allows users to select a file to upload.

<!DOCTYPE **html**>  
<**html**>  
<**body**>  
  
<**form action="upload.php" method="post" enctype="multipart/form-data"**>  
 Select image to upload:  
 <**input type="file" name="fileToUpload" id="fileToUpload"**>  
 <**input type="submit" value="Upload Image" name="submit"**>  
</**form**>  
  
</**body**>  
</**html**>

Next create the PHP file to process and upload.

**<?php**$target\_dir = **"uploads/"**;  
$target\_file = $target\_dir . *basename*($\_FILES[**"fileToUpload"**][**"name"**]);  
$uploadOk = 1;

$imageFileType = *strtolower*(*pathinfo*($target\_file,***PATHINFO\_EXTENSION***));

*// Check if image file is a actual image or fake image***if**(**isset**($\_POST[**"submit"**])) {  
 $check = *getimagesize*($\_FILES[**"fileToUpload"**][**"tmp\_name"**]);  
 **if**($check !== **false**) {  
 **echo "File is an image - "** . $check[**"mime"**] . **"."**;  
 $uploadOk = 1;  
 } **else** {  
 **echo "File is not an image."**;  
 $uploadOk = 0;  
 }  
}

*// Check if file already exists***if** (*file\_exists*($target\_file)) {  
 **echo "Sorry, file already exists."**;  
 $uploadOk = 0;  
}

*// Check file size***if** ($\_FILES[**"fileToUpload"**][**"size"**] > 500000) {  
 **echo "Sorry, your file is too large."**;  
 $uploadOk = 0;  
}

*// Allow certain file formats***if**($imageFileType != **"jpg"** && $imageFileType != **"png"** && $imageFileType != **"jpeg"** && $imageFileType != **"gif"** ) {  
 **echo "Sorry, only JPG, JPEG, PNG & GIF files are allowed."**;  
 $uploadOk = 0;  
}

*// Check if $uploadOk is set to 0 by an error***if** ($uploadOk == 0) {  
 **echo "Sorry, your file was not uploaded."**;  
*// if everything is ok, try to upload file*}

**else** {  
 **if** (*move\_uploaded\_file*($\_FILES[**"fileToUpload"**][**"tmp\_name"**], $target\_file)) {  
 **echo "The file "**. *basename*( $\_FILES[**"fileToUpload"**][**"name"**]). **" has been uploaded."**;  
 }

**else** {  
 **echo "Sorry, there was an error uploading your file."**;  
 }  
}  
**?>**

* $target\_dir = "uploads/" - specifies the directory where the file is going to be placed
* $target\_file specifies the path of the file to be uploaded
* $uploadOk=1 is not used yet
* $imageFileType holds the file extension of the file (in lower case)
* Next, check if the image file is an actual image or a fake image

# fopen() and fclose()

fopen() is the basis for file manipulation. It opens a file in a certain mode that you specify and returns a handle. Using this handle you can read or write to the file, before closing it with the fclose() function.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| filename | Required. Specifies the file or URL to open |
| mode | Required. Specifies the type of access you require to the file/stream.  Possible values:   * "r" (Read only. Starts at the beginning of the file) * "r+" (Read/Write. Starts at the beginning of the file) * "w" (Write only. Opens and clears the contents of file; or creates a new file if it doesn't exist) * "w+" (Read/Write. Opens and clears the contents of file; or creates a new file if it doesn't exist) * "a" (Write only. Opens and writes to the end of the file or creates a new file if it doesn't exist) * "a+" (Read/Write. Preserves file content by writing to the end of the file) * "x" (Write only. Creates a new file. Returns FALSE and an error if file already exists) * "x+" (Read/Write. Creates a new file. Returns FALSE and an error if file already exists) |
| include\_path | Optional. Set this parameter to '1' if you want to search for the file in the include\_path (in php.ini) as well |
| context | Optional. Specifies the context of the file handle. Context is a set of options that can modify the behavior of a stream |

Syntax: fopen(filename,mode,include\_path,context)

Syntax: fclose(file)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| file | Required. Specifies the file to close |

Sample Code

**<?php**$handle = *fopen*(**'data.txt'**, **'r'**); *// Open the file for reading  
fclose*($handle); *// Close the file***?>**

In the example above you can see the file is opened for reading by specifying 'r' as the mode. For a full list of all the modes available to fopen(), you can look at the PHP Manual page.

Opening and closing the file is all well and good, but to perform useful operations, you need to know about fread() and fwrite().

When a PHP script finishes executing, all open files are automatically closed. So although it is not strictly necessary to close a file after opening it, it is considered good programming practice to do so.

# Reading

Reading can be done in a number of ways. If you just want all the contents of a file available to work with, you can use the file\_get\_contents() function. If you want each line of the file in an array, you can use the file() command. For total control over reading from files, fread() can be used.

These functions are usually interchangeable and each can be used to perform each other's function. The first two do not require that you first open the file with fopen() or then close it with fclose(). These are good for quick, one-time file operations. If you plan on performing multiple operations on a file it is best to use fopen() in conjunction with fread(), fwrite() and fclose() as it is more efficient.

Syntax: file\_get\_contents(path,include\_path,context,start,max\_length)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| path | Required. Specifies the file to read |
| include\_path | Optional. Set this parameter to '1' if you want to search for the file in the include\_path (in php.ini) as well |
| context | Optional. Specifies the context of the file handle. Context is a set of options that can modify the behavior of a stream. Can be skipped by using NULL. |
| start | Optional. Specifies where in the file to start reading. This parameter was added in PHP 5.1 |
| max\_length | Optional. Specifies how many bytes to read. This parameter was added in PHP 5.1 |

An example of using file\_get\_contents()

**Code:**

<?php

$contents = file\_get\_contents('data.txt');

**echo** $contents;

?>

**Output:**

I am the contents of data.txt

This function reads the entire file into a string and from then on you can manipulate it as you would any string.

An example of using file()

**Code:**

<?php

$lines = file('data.txt');

**foreach**($lines **as** $Key => $line) {

$lineNum = $Key + 1;

**echo** "Line **$lineNum**: **$line**";

}

?>

**Output:**

Line 1: I am the first line of file

Line 2: I am the second line the of the file

Line 3: If I said I was the fourth line of the file, I'd be lying

Syntax: fread(file,length)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| file | Required. Specifies the open file to read from |
| length | Required. Specifies the maximum number of bytes to read |

This function reads the entire file into an array. Each item in the array corresponds to a line in the file.

An example of using fread()

**Code:**

<?php

$handle = fopen('data.txt', 'r');

$string = fread($handle, 64);

fclose($handle);

**echo** $string;

?>

**Output:**

I am the first 64 bytes of data.txt (if it was ASCII encoded). I

This function can read up to the specified number of bytes from the file and return it as a string. For the most part, the first two functions will be preferable, but there are occasions when this function is needed.

As you can see, with these three functions you are able to easily read data from a file into a form that is convenient to work with. The next part shows how these functions can be used to do the jobs of the others, but this is optional. You may skip it and move onto the writing section, if you are not interested.

<?php

$file = 'data.txt';

**function** detectLineEndings($contents) {

**if**(**false** !== strpos($contents, "**\r\n**")) **return** "**\r\n**";

**else** **if**(**false** !== strpos($contents, "**\r**")) **return** "**\r**";

**else** **return** "**\n**";

}

*/\* This is equivalent to file\_get\_contents($file), but is less efficient \*/*

$handle = fopen($file, 'r');

$contents = fread($handle, filesize($file));

fclose($handle);

*/\* This is equivalent to file($file), but requires you to check for the line-ending type. Windows systems use \r\n, Macintosh \r and Unix \n. File($file) will automatically detect line-endings whereas fread/file\_get\_contents won't \*/*

$lineEnding = detectLineEndings($contents);

$contents = file\_get\_contents($file);

$lines = explode($lineEnding, $contents);

*/\* This is also equivalent to file\_get\_contents($file) \*/*

$lines = file($file);

$contents = implode("**\n**", $lines);

*/\* This is equivalent to fread($file, 64), if the file is ASCII encoded \*/*

$contents = file\_get\_contents($file);

$string = substr($contents, 0, 64);

?>

**Writing**

Writing to a file is done by using the fwrite() function in conjunction with fopen() and fclose(). As you can see, there aren't as many options for writing to a file as there are for reading from one. However, PHP 5 introduces the function file\_put\_contents() that simplifies the writing process somewhat. This function will be discussed later in the PHP 5 section, as it is fairly self-explanatory and does not require discussion here.

Syntax: file\_put\_contents(file,data,mode,context)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| file | Required. Specifies the file to write to. If the file does not exist, this function will create one |
| data | Required. The data to write to the file. Can be a string, an array or a data stream |
| mode | Optional. Specifies how to open/write to the file. Possible values:   * FILE\_USE\_INCLUDE\_PATH * FILE\_APPEND * LOCK\_EX |
| context | Optional. Specifies the context of the file handle. Context is a set of options that can  modify the behavior of a stream. |

Sample Code:

**<?php  
echo** *file\_get\_contents*(**"test.txt"**);  
**?>**

The extra options for writing don't come from the amount of functions, but from the modes available for opening the file. There are three different modes you can supply to the fopen() function, if you wish to write to a file. One mode, 'w', wipes the entire contents of the file, so anything you then write to the file will fully replace what was there before. The second mode, 'a', appends stuff to the file so anything you write to the file will appear just after the original contents of the file. The final mode 'x' only works for non-existent files. All three writing modes will attempt to create the file, if it doesn't exist whereas the 'r' mode will not.

An example of using the 'w' mode

**Code:**

<?php

*// Open the file and delete its contents*

$handle = fopen('data.txt', 'w');

$data = "I am new content**\n**spread across**\n**several lines.";

fwrite($handle, $data);

fclose($handle);

**echo** file\_get\_contents('data.txt');

?>

**Output:**

I am new content

spread across

several lines.

An example of using the 'a' mode

**Code:**

<?php

$handle = fopen('data.txt', 'a'); *// Open the file for appending*

$data = "**\n\n**I am new content.";

fwrite($handle, $data);

fclose($handle);

**echo** file\_get\_contents('data.txt');

?>

**Output:**

I am the original content.

I am new content.

An example of using the 'x' mode

**Code:**

<?php

$handle = fopen('newfile.txt', 'x'); *// Open the file only, if it doesn't exist*

$data = "I am this file's first ever content!";

fwrite($handle, $data);

fclose($handle);

**echo** file\_get\_contents('newfile.txt');

?>

**Output:**

I am this file's first ever content!

Of the three modes shown above, 'w' and 'a' are used the most, but the writing process is essentially the same for all the modes.

# Reading *and* Writing

If you want to use [fopen()](http://php.net/fopen) to open a file for both reading and writing all you need to do is put a '+' on the end of the mode. For example, reading from a file requires the 'r' mode. If you want to read and write to/from that file you need to use 'r+' as a mode. Similarly, you can read and write to/from a file using the 'w+' mode. However, this will also truncate the file to zero length. For a better description visit the [fopen()](http://php.net/fopen) page that has a very useful table describing all the modes available.

# Error Checking

Error checking is important for any sort of programming, but when working with files in PHP it is especially important. This need for error checking arises mainly from the filesystem the files are on. The majority of webservers today are Linux-based and so, if you are using PHP to develop web-applications, you have to account for file permissions. In some cases, PHP may not have permission to read the file and so, if you've written code to read a particular file, it will result in an ugly error. More likely is that PHP doesn't have permission to write to a file and that will again result in ugly errors. Also, the file's existence is (somewhat obviously) important. When attempting to read a file, you must make sure the file exists first. On the other side of that, if you're attempting to create and then write to a file using the 'x' mode, then you must make sure the file doesn't exist first.

In short, when writing code to work with files, always assume the worst. Assume the file doesn't exist and you don't have permission to read from/write to it. In most cases this means you have to tell the users that, in order for the script to work, they need to adjust those file permissions so that PHP can create files and read from/write to them, but it also means that your script can adjust and perform an alternative operation.

There are two main ways of error checking. The first is by using the '[@](http://uk.php.net/manual/en/language.operators.errorcontrol.php)' operator to suppress any errors when working with the file and then checking, if the result is false or not. The second method involves using more functions like [file\_exists()](http://php.net/file_exists), [is\_readable()](http://php.net/is_readable)and [is\_writeable()](http://php.net/is_writeable).

Examples of using the '@' operator

<?php

$handle = @ fopen('data.txt', 'r');

**if**(!$handle) {

**echo** 'PHP does not have permission to read this file or the file in question doesn\'t exist.';

} **else** {

$string = fread($handle, 64);

fclose($handle);

}

$handle = @ fopen('data.txt', 'w'); *// The same applies for 'a'*

**if**(!$handle) {

**echo** 'PHP either does not have permission to write to this file or it does not have permission to create this file in the current directory.';

} **else** {

fwrite($handle, 'I can has content?');

fclose($handle);

}

$handle = @ fopen('data.txt', 'x');

**if**(!$handle) {

**echo** 'Either this file exists or PHP does not have permission to

create this file in the current directory.';

} **else** {

fwrite($handle, 'I can has content?');

fclose($handle);}

?>

As you can see, the '@' operator is used mainly when working with the *fopen()* function. It can also be used in other cases, but is generally less efficient.

Examples of using specific checking functions

<?php

$file = 'data.txt';

**if**(!file\_exists($file)) {

*// No point in reading since there is no content*

$contents = '';

*// But might want to create the file instead*

$handle = @ fopen($file, 'x'); *// Still need to error-check ;)*

**if**(!$handle) {

**echo** 'PHP does not have permission to create a file in the current directory.';

} **else** {

fwrite($handle, 'Default data');

fclose($handle);

}

} **else** {

*// The file does exist so we can try to read its contents*

**if**(is\_readable($file)) {

$contents = file\_get\_contents($file);

} **else** {

**echo** 'PHP does not have permission to read that file.';

}

}

**if**(file\_exists($file) && is\_writeable($file)) {

$handle = fopen($file, 'w');

fwrite($handle, 'I can has content?');

fclose($handle);

}

?>

You can see by that last example that error-checking makes your code very robust. It allows it to be prepared for most situations and behave accordingly, which is an essential aspect of any program or script.

# Line-endings

Line-endings were mentioned briefly in the final example in the 'Reading' section of this section and it is important to be aware of them when working with files. When reading from a text file, it is important to know what types of line-endings that file contains. 'Line-endings' are special characters that try to tell a program to display a new line. For example, Notepad will only move a piece of text to a new line, if it finds "\r\n" just before the new line (it will also display new lines, if you put word wrap on).

If someone writes a text file on a Windows system, the chances are that each line will end with "\r\n". Similarly, if they write the file on a Classic Macintosh (Mac OS 9 and under) system, each line will probably end with "\r". Finally, if they write the file on a Unix-based system (Mac OS X and GNU/Linux), each line will probably end with "\n".

Why is this important? Well, when you read a file into a string with file\_get\_contents(), the string will be one long line with those line-endings all over the place. Sometimes they will get in the way of things you want to do with the string so you can remove them with:

<?php

$string = str\_replace(**array**("**\n**", "**\r**"), '', $string);

?>

Other times you may need to know what kind of line-ending is being used throughout the text in order to be consistent with any new text you add. Luckily, in 99% of cases, the line-endings will never change type throughout the text so the custom function 'detectLineEndings' can be used as a quick way of checking:

<?php

**function** detectLineEndings($string) {

**if**(**false** !== strpos($string, "**\r\n**")) **return** "**\r\n**";

**else** **if**(**false** !== strpos($string, "**\r**")) **return** "**\r**";

**else** **return** "**\n**";

}

?>

Most of the time though, it is just sufficient to be aware of their existence within the text so you can adjust your script to cope properly.

# Binary-safe

So far, all of the text seen in this chapter has been assumed to be encoded in some form of plaintext encoding such as UTF-8 or ASCII. Files do not have to be in this format, however, and in fact there exist a huge number of formats that aren't (such as pictures or executables). If you want to work with these files you have to ensure that the functions, you are using are 'binary-safe'. Previously you would have to add 'b' to the end of the modes you used to tell PHP to treat the file as a binary file. Failing to do so would give unexpected results and generally 'weird-looking' data.

Since about PHP 4.3, this is no longer necessary as PHP will automatically detect, if it needs to open the file as a text file or a binary file and so you can still follow most of the examples shown here.

Working with binary data is a lot different to working with plaintext strings and characters and involves many more functions that are beyond the scope of this chapter. However, it is important you know about these differences.

# Serialization

Serialization is a technique used by programmers to preserve their working data in a format that can later be restored to its previous form. In simple cases this means converting a normal variable such as an array into a string and then storing it somewhere. That data can then be unserialized and the programmer will be able to work with the array once again.

There is a whole chapter devoted to Serialization in this book as it is a useful technique to know how to use effectively. It is mentioned here as one of the primary uses of serialization to store data on plain files when a database is not available. It is also used to store the state of a script and to cache data for quicker access later, and files are one of the preferred media for this storage.

In PHP, serialization is very easy to perform through use of the serialize() and unserialize() functions. Here follows an example of serialization used in conjunction with file functions.

An example of storing user details in a file so that they can be easily retrieved later.

**Code:**

<?php

*/\* This part of the script saves the data to a file \*/*

$data = **array**(

'id' => 114,

'first name' => 'Foo',

'last name' => 'Bartholomew',

'age' => 21,

'country' => 'England'

);

$string = serialize($data);

$handle = fopen('data.dat', 'w');

fwrite($handle, $string);

fclose($handle);

*/\* Then, later on, we retrieve the data from the file and output it \*/*

$string = file\_get\_contents('data.dat');

$data = unserialize($string);

$output = '';

**foreach**($data **as** $key => $datum) {

$field = ucwords($key);

$output .= "**$field**: **$datum\n**";

}

**echo** $output

?>

**Output:**

Id: 114

First Name: Foo

Last Name: Bartholomew

Age: 21

Country: England

# PHP 5

There is one particular function specific to files that was introduced in PHP 5. That was the file\_put\_contents() function. It offers an alternative method of writing to files that does not exist in PHP 4. To see how it differs, it is easiest to just look at an example.

Examples showing writing to a file in PHP 4 and the equivalent in PHP 5 with the file\_put\_contents() function

<?php

$file = 'data.txt';

$content = 'New content.';

*// PHP 4, overwrite entire file with data*

$handle = fopen($file, 'w');

fwrite($handle, $content);

fclose($handle);

*// PHP 5*

file\_put\_contents($file, $content);

*// PHP 4, append to a file*

$handle = fopen($file, 'a');

fwrite($handle, $content);

fclose($handle);

*// PHP 5*

file\_put\_contents($file, $content, FILE\_APPEND);

?>

*file\_put\_contents()* will also attempt to create the file, if it doesn't exist and it is binary-safe. There is no equivalent of the 'x' mode for *file\_get\_contents()*.

file\_put\_contents() is almost always preferable over the fopen() method except when performing multiple operations on the same file. It is more preferable to use it for writing than file\_get\_contents() is for reading and for this reason, a function is provided here to emulate the behavior of file\_put\_contents() for PHP 4:

<?php

**if**(!function\_exists('file\_put\_contents')) {

**function** file\_put\_contents($file, $data, $append = **false**) {

**if**(!$append) $mode = 'w';

**else** $mode = 'a';

$handle = @fopen($file, $mode);

**if**(!$handle) **return** **false**;

$bytes = fwrite($handle, $data);

fclose($handle);

**return** $bytes;

}

}

?>

The file\_put\_contents() function can be used to add content to the end of a file or replace an existing file. The function fwrite() can also be used to add content to file. The fwrite() return on the amount of data written. With the fwrite() function, the developer will have to use fopen() to create a handle to the file first.

**<?php**$file = *fopen*(**"test.txt"**,**"w"**);  
**echo** *fwrite*($file,**"Hello World. Testing!"**);  
*fclose*($file);  
**?>**

# Reading In Files

When working with large files, it’s is recommended to read the file in sections. It will improve overall system performance, since processing large files all at once requires sufficient amount of system memory. By using a file pointer that moves with each iteration of a loop to read in the file, allow your code to read in chucks of data at a time. The functions fgetc() and fread() do just that. Each time the getc() and fread() function is called, the file pointer will automatically move to the either to next character or the next available character till the end of file is detected. To determine if your code has reached the end of file use the feof() function, which returns TRUE when a file pointer reaches the end of a file.

Syntax: fread(file,length)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| file | Required. Specifies the open file to read from |
| length | Required. Specifies the maximum number of bytes to read |

**<?php**$file = *fopen*(**"test.txt"**,**"r"**);  
*fread*($file,**"10"**);  
*fclose*($file);  
**?>**

Syntax: fgetc(file)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| file | Required. Specifies the file to check |

**<?php**$file = *fopen*(**"test2.txt"**,**"r"**);  
**echo** *fgetc*($file);  
*fclose*($file);  
**?>**