Working with Files in PHP

By Matt Zandstra

Date: Oct 11, 2002

Sample Chapter is provided courtesy of Sams.

Return to the article

Most programming languages provide mechanisms for reading files and saving data to disk. This lesson examines PHP's functions for file handling.

Testing, reading, and writing to files are staple activities for any full-featured programming language. PHP is no exception, providing you with functions that make the process straightforward. In this hour, you will learn

- How to include files in your documents
- How to test files and directories
- How to open a file before working with it
- How to read data from files
- How to write or append to a file
- How to lock a file
- How to work with directories

Including Files with `include()`

The `include()` statement enables you to incorporate files into your PHP documents. PHP code in these files can be executed as if it were part of the main document. This can be useful for including library code in multiple pages.

Having created a killer function, your only option until now would have been to paste it into every document that needs to use it. Of course, if you discover a bug, or want to add a feature, you will have to find every page that uses the function to make the change. The `include()` statement can save you from this chore. You can add the function to a single document and, at runtime, read this into any page that needs it. The `include()` statement requires a single argument, a relative path to the file to be included. Listing 10.1 creates a simple PHP script that uses `include()` to incorporate and output the contents of a file.

Listing 10.1 Using `include()`

```
 1: <html>
 2: <head>
 3: <title>Listing 10.1 Using include()</title>
 4: </head>
 5: <body>
 6: <?php
 7: include("listing10.2.php");
 8: ?>
 9: </body>
10: </html>
```

The include() statement in Listing 10.1 incorporates the document listing10.2.php, the contents of which you can see in Listing 10.2. When run, Listing 10.1 outputs the string "I have been included!!", which may seem strange, given that we have included plain text within a block of PHP code. Shouldn't this cause an error? In fact, the contents of an included file are displayed as HTML by default. If you want to execute PHP code in an included file, you must enclose it in PHP start and end tags. In Listings 10.3 and 10.4, we amend the previous example so that code is executed in the included file.

Listing 10.2 The File Included in Listing 10.1

```
1: I have been included!!
```

Listing 10.3 Using the `include()` statement to Execute PHP in Another File

```
 1: <html>
 2: <head>
 3: <title>Listing 10.3 Using include to execute PHP in another file</title>
 4: </head>
 5: <body>
 6: <?php
 7: include("listing10.4.php");
 8: ?>
 9: </body>
10: </html>
```

Listing 10.4 An Include File Containing PHP Code

```
1: <?php
2: print "I have been included!!<BR>";
3: print "But now I can add up... 4 + 4 = ".(4 + 4);
4: ?>
```

Returning a Value From an Included Document

Included files in PHP4 can return a value in the same way as functions do. As in a function, using the `return` statement ends the execution of code within the included file. Additionally, no further HTML will be included. In Listings 10.5 and 10.6, we include a file, assigning its return value to a variable.

Listing 10.5 Using `include()` to Execute PHP and Assign the Return Value

```
 1: <html>
 2: <head>
 3: <title>Listing 10.5 Using include() to execute PHP and assign the return value</title>
 4: </head>
 5: <body>
 6: <?php
 7: $addResult = include("listing10.6.php");
 8: print "The include file returned $addResult";
 9: ?>
10: </body>
11: </html>
```

Listing 10.6 An Include File that Returns a Value

```
1: <?php
2: $retval = ( 4 + 4 );
3: return $retval;
4: ?>
5: This HTML should never be displayed because it comes after a return statement!
```

NOTE

Returning values from included files would only work in PHP3 if the `return` statement was contained in a function. The code in Listing 10.6 would cause an error.

Using `include()` Within Control Structures

You can use an `include()` statement in a conditional statement, and the referenced file will only be read if the condition is met. The `include()` statement in the following fragment will never be called, for example

```
$test = false;
if ( $test ) {
  include( "a_file.txt" ); // won't be included
}
```

If you use an include() statement within a loop, it will be replaced with the contents of the referenced file each time the include() statement is called. This content will be executed for every call. Listing 10.7 illustrates this by using an include() statement in a for loop. The include() statement references a different file for each iteration.

Listing 10.7 Using `include()` Within a Loop

```
 1: <html>
 2: <head>
 3: <title>Listing 10.7 Using include() within a loop</title>
 4: </head>
 5: <body>
 6: <?php
 7: for ( $x = 1; $x<=3; $x++ ) {
 8:  $incfile = "incfile$x".".txt";
 9:  print "Attempting include $incfile<br>";
10:  include( "$incfile" );
11:  print "<p>";
12: }
13: ?>
14: </body>
15: </html>
```

When Listing 10.7 is run, it includes the content of three different files, "incfile1.txt", "incfile2.txt", and "incfile3.txt". Assuming that each of these files simply contains a confirmation of its own name, the output should look something like this:

```
Attempting include incfile1.txt
This is incfile1.txt

Attempting include incfile2.txt
This is incfile2.txt

Attempting include incfile3.txt
This is incfile3.txt
```

`include_once()`

One of the problems caused by using multiple libraries within your code is the danger of calling `include()` twice on the same file. This can occur in larger projects when different library files call `include()` on a common file. Including the same file twice will often result in the repeated declaration of functions and classes, thereby causing the PHP engine great unhappiness.

The situation is saved by the `include_once()` statement. `include_once()` requires the path to an include file and will behave the same way as `include()` the first time it is called. If `include_once()` is called again for the same file during script execution, however, the file will *not* be included again.

This makes `include_once()` an excellent tool for the creation of reusable code libraries.

The `include_path` directive

Using `include()` and `include_once()` to access libraries can increase the flexibility and reusability of your projects. However there are still headaches to overcome. Portability in particular can suffer if you hardcode the paths to included files. Imagine that you create a 'lib' directory and reference it throughout your project:

```
include_once("/home/user/bob/htdocs/project4/lib/mylib.inc.php");
```

When you come to move your project to a new server you may find that you have to change a hundred or more include paths. You can escape this fate by setting the include_path directive in your php.ini file.

include_path .:/home/user/bob/htdocs/project4/lib/

include_path can include as many directories as you want separated by colons (semicolons in Windows). You can then reference your library file only by its name

include_once("mylib.inc.php");

When you move your project you will only need to change the include_path directive.

NOTE

PHP4 has a `require()` statement, which performs a similar function to `include()`. There is also a `require_once()` statement.

`require()` is executed regardless of a script's flow, and should not therefore be used as part conditional or loop structures.

A file included as a result of a `require()` statement cannot return a value.

Testing Files

Before you work with a file or directory, it is often a good idea to learn more about it. PHP4 provides many functions that help you to discover information about files on your system. This section briefly covers some of the most useful.

Checking for Existence with `file_exists()`

You can test for the existence of a file with the `file_exists()` function. This requires a string representing an absolute or relative path to a file that may or may not be there. If the file is found, it returns `true`; otherwise, it returns `false`.

```
if ( file_exists("test.txt") )
  print "The file exists!";
```

A File or a Directory?

You can confirm that the entity you are testing is a file, as opposed to a directory, with the `is_file()` function. `is_file()` requires the file path and returns a Boolean value:

```
if ( is_file( "test.txt" ) )
  print "test.txt is a file!";
```

Conversely, you might want to check that the entity you are testing is a directory. You can do this with the is_dir() function. is_dir() requires the path to the directory and returns a Boolean value:

```
if ( is_dir( "/tmp" ) )
  print "/tmp is a directory";
```

Checking the Status of a File

When you know that a file exists, and it is what you expect it to be, you can then find out some things that you can do with it. Typically, you might want to read, write to, or execute a file. PHP can help you with all of these.

`is_readable()` tells you whether you can read a file. On UNIX systems, you may be able to see a file but still be barred from reading its contents. `is_readable()` accepts the file path as a string and returns a Boolean value:

```
if ( is_readable( "test.txt" ) )
  print "test.txt is readable";
```

is_writable() tells you whether you can write to a file. Once again it requires the file path and returns a Boolean value:

```
if ( is_writable( "test.txt" ) )
  print "test.txt is writable";
```

is_executable() tells you whether you can run a file, relying on either the file's permissions or its extension depending on your platform. It accepts the file path and returns a Boolean value:

```
if ( is_executable( "test.txt" )
  print "test.txt is executable";
```

Determining File Size with `filesize()`

Given the path to a file, `filesize()` attempts to determine and return its size in bytes. It returns `false` if it encounters problems.

```
print "The size of test.txt is.. ";
print filesize( "test.txt" );
```

Getting Date Information About a File

Sometimes you will need to know when a file was last written to or accessed. PHP provides several functions that can provide this information.

You can find out when a file was last accessed with `fileatime()`. This function requires the file path and returns the date that the file was last accessed. To access a file means either to read or write to it. Dates are returned from all these functions in UNIX epoch format. That is, the number of seconds since 1 January 1970. In our examples, we use the `date()` function to translate this into human readable form. You learn more about date functions in Hour 15, "Working with Dates and Times."

```
$atime = fileatime( "test.txt" );
print "test.txt was last accessed on ";
print date("D d M Y g:i A", $atime);
// Sample output: Thu 13 Jan 2000 2:26 PM
```

You can discover the modification date of a file with the function filemtime(), which requires the file path and returns the date in UNIX epoch format.

To modify a file means to change its contents in some way.

```
$mtime = filemtime( "test.txt" );
print "test.txt was last modified on ";
print date("D d M Y g:i A", $mtime);
// Sample output: Thu 13 Jan 2000 2:26 PM]
```

PHP also allows you to test the change time of a document with the filectime() function. On Unix systems, the change time is set when a file's contents are modified or changes are made to its permissions or ownership. On other platforms, the filectime() returns the creation date.

```
$ctime = filectime( "test.txt" );
print "test.txt was last changed on ";
print date("D d M Y g:i A", $ctime);
// Sample output: Thu 13 Jan 2000 2:26 PM]
```

Creating a Function that Performs Multiple File Tests

Listing 10.8 creates a function that brings the file test functions we have looked at together into one script.

Listing 10.8 A Function to Output the Results of Multiple File Tests

```
 1: <html>
 2: <head>
 3: <title>Listing 10.8 A function to output the results of multiple file tests</title>
 4: </head>
 5: <body>
 6: <?php
 7: $file = "test.txt";
 8: outputFileTestInfo( $file );
 9:
10: function outputFileTestInfo( $f ) {
11:   if ( ! file_exists( $f ) ) {
12:     print "$f does not exist<BR>";
13:     return;
14:   }
15:   print "$f is ".(is_file( $f )?"":"not ")."a file<br>";
16:   print "$f is ".(is_dir( $f )?"":"not ")."a directory<br>";
17:   print "$f is ".(is_readable( $f )?"":"not ")."readable<br>";
18:   print "$f is ".(is_writable( $f )?"":"not ")."writable<br>";
19:   print "$f is ".(is_executable( $f )?"":"not ")."executable<br>";
20:   print "$f is ".(filesize($f))." bytes<br>";
21:   print "$f was accessed on ".date( "D d M Y g:i A", fileatime( $f ) )."<br>";
22:   print "$f was modified on ".date( "D d M Y g:i A", filemtime( $f ) )."<br>";
23:   print "$f was changed on ".date( "D d M Y g:i A", filectime( $f ) )."<br>";
24: }
25:
26: ?>
27: </body>
28: </html>
```

Notice that we have used the ternary operator as a compact way of working with some of these tests. Let's look at one of these, found in line 15, in more detail:

```
print "$f is ".(is_file( $f )?"":"not ")."a file<br>";
```

We use the is_file() function as the right-hand expression of the ternary operator. If this returns true, an empty string is returned. Otherwise, the string "not " is returned. The return value of the ternary expression is added to the string to be printed with concatenation operators. This statement could be made clearer but less compact, as follows:

```
$is_it = is_file( $f )?"":"not ";
print "$f is $isit a file";
```

We could, of course, be even clearer with an if statement, but imagine how large the function would become if we had used the following:

```
if ( is_file( $f ) )
  print "$fi is a file<br>";
else
  print "$f is not a file<br>";
```

Because the result of these three approaches is the same, the approach you take becomes broadly a matter of preference.

Creating and Deleting Files

If a file does not yet exist, you can create one with the `touch()` function. Given a string representing a file path, `touch()` attempts to create an

empty file of that name. If the file already exists, the contents are not disturbed, but the modification date is updated to the time at which the function executed.

```
touch("myfile.txt");
```

You can remove an existing file with the unlink() function. Once again, unlink() accepts a file path:

```
unlink("myfile.txt");
```

All functions that create, delete, read, write, or modify files on Unix systems require that the correct file or directory permissions are set.

Opening a File for Writing, Reading, or Appending

Before you can work with a file, you must first open it for reading, writing, or both. PHP provides the `fopen()` function for this. `fopen()` requires a string containing the file path, followed by a string containing the mode in which the file is to be opened. The most common modes are read (`'r'`), write (`'w'`), and append (`'a'`). `fopen()` returns a file resource you will later use to work with the open file. To open a file for reading, you would use the following:

```
$fp = fopen( "test.txt", 'r' );
```

You would use the following to open a file for writing:

```
$fp = fopen( "test.txt", 'w' );
```

To open a file for appending (that is, to add data to the end of a file), you would use this:

```
$fp = fopen( "test.txt", 'a' );
```

fopen() returns false if the file cannot be opened for any reason. It is a good idea, therefore, to test the function's return value before proceeding to work with it. You can do this with an if statement:

```
if ( $fp = fopen( "test.txt", "w" ) ) {
  // do something with $fp
}
```

Or you can use a logical operator to end execution if an essential file can't be opened:

```
( $fp = fopen( "test.txt", "w" ) ) or die ("Couldn't open file, sorry");
```

If the fopen() function returns true, the rest of the expression won't be parsed, and the die() function (which writes a message to the browser and ends the script) will never be reached. Otherwise, the right-hand side of the or operator will be parsed, and the die() function will be called.

Assuming that all is well and you go on to work with your open file, you should remember to close it when you have finished. You can do this by calling `fclose()`, which requires the file resource returned from a successful `fopen()` call as its argument:

```
fclose( $fp );
```

Reading from Files

PHP provides a number of functions for reading data from files. These enable you to read by the byte, the line, or even the character.

Reading Lines from a File with `fgets()` and `feof()`

After you have opened a file for reading, you will often need to access it line by line. To read a line from an open file, you can use `fgets()`, which requires the file resource returned from `fopen()` as an argument. You must also pass it an integer as a second argument. This specifies the number of bytes the function should read if it doesn't first encounter a line end or the end of the file. The `fgets()` function reads the file until it reaches a newline character (`"\n"`), the number of bytes specified in the length argument, or the end of the file.

```
$line = fgets( $fp, 1024 ); // where $fp is the file resource returned by fopen()
```

Although you can read lines with fgets(), you need some way of telling when you have reached the end of the file. The feof() function does this, returning true when the end of the file has been reached and false otherwise. Once again this function requires a file resource as its argument:

```
feof( $fp ); // where $fp is the file resource returned by fopen()
```

You now have enough information to read a file line by line, as shown in Listing 10.9.

Listing 10.9 Opening and Reading a File Line by Line

```
 1: <html>
 2: <head>
 3: <title>Listing 10.9 Opening and reading a file line by line</title>
 4: </head>
 5: <body>
 6: <?php
 7: $filename = "test.txt";
 8: $fp = fopen( $filename, "r" ) or die("Couldn't open $filename");
 9: while ( ! feof( $fp ) ) {
10:   $line = fgets( $fp, 1024 );
11:   print "$line<br>";
12: }
13: ?>
14: </body>
15: </html>
```

We call fopen() on line 8 with the name of the file that we want to read, using the or operator to ensure that script execution ends if the file cannot be read. This usually occurs if the file does not exist, or (on a Unix system) if the file's permissions won't allow the script read access to the file. The actual reading takes place in the while statement on line 9. The while statement's test expression calls feof() for each iteration, ending the loop when it returns true. In other words, the loop continues until the end of the file is reached. Within the code block, we use fgets() on line 10 to extract a line (or 1024 bytes) of the file. We assign the result to $line and then print it to the browser on line 11, appending a <BR> tag for the sake of readability.

Reading Arbitrary Amounts of Data from a File with `fread()`

Rather than reading text by the line, you can choose to read a file in arbitrarily defined chunks. The `fread()` function accepts a file `resource` as an argument, as well as the number of bytes you want to read. It returns the amount of data you have requested unless the end of the file is reached first.

```
$chunk = fread( $fp, 16 );
```

Listing 10.10 amends our previous example so that it reads data in chunks of 16 bytes rather than by the line.

Listing 10.10 Reading a File with `fread()`

```
 1: <html>
 2: <head>
 3: <title>Listing 10.10 Reading a file with fread()</title>
 4: </head>
 5: <body>
 6: <?php
 7: $filename = "test.txt";
 8: $fp = fopen( $filename, "r" ) or die("Couldn't open $filename");
 9: while ( ! feof( $fp ) ) {
10:   $chunk = fread( $fp, 16 );
11:   print "$chunk<br>";
12: }
13: ?>
14: </body>
15: </html>
```

Although fread() allows you to define the amount of data acquired from a file, it won't let you decide the position from which the acquisition begins. You can set this manually with the fseek() function. fseek() enables you to change your current position within a file. It requires a file resource and an integer representing the offset from the start of the file (in bytes) to which you want to jump:

```
fseek( $fp, 64 );
```

Listing 10.11 uses fseek() and fread() to output the second half of a file to the browser.

Listing 10.11 Moving Around a File with `fseek()`

```
 1: <html>
 2: <head>
 3: <title>Listing 10.11 Moving around a file with fseek()</title>
 4: </head>
 5: <body>
 6: <?php
 7: $filename = "test.txt";
 8: $fp = fopen( $filename, "r" ) or die("Couldn't open $filename");
 9: $fsize = filesize($filename);
10: $halfway = (int)( $fsize / 2 );
11: print "Halfway point: $halfway <BR>\n";
12: fseek( $fp, $halfway );
13: $chunk = fread( $fp, ($fsize - $halfway) );
```

```
14: print $chunk;
15: ?>
16: </body>
17: </html>
```

We calculate the halfway point of our file by dividing the return value of filesize() by 2 on line 10. We can then use this as the second argument to fseek() on line 12, jumping to the halfway point. Finally, we call fread() on line 13 to extract the second half of the file, printing the result to the browser.

Reading Characters from a File with `fgetc()`

`fgetc()` is similar to `fgets()` except that it returns only a single character from a file every time it is called. Because a character is always 1 byte in size, `fgetc()` doesn't require a length argument. You simply need to pass it a file resource:

```
$char = fgetc( $fp );
```

Listing 10.12 creates a loop that reads the file "test.txt" a character at a time, outputting each character to the browser on its own line.

Listing 10.12 Moving Around a File with `fseek()`

```
 1: <html>
 2: <head>
 3: <title>Listing 10.12</title>
 4: </head>
 5: <body>
 6: <?php
 7: $filename = "test.txt";
 8: $fp = fopen( $filename, "r" ) or die("Couldn't open $filename");
 9: while ( ! feof( $fp ) ) {
10:  $char = fgetc( $fp );
11:  print "$char<BR>";
12: }
13: ?>
14: </body>
15: </html>
```

Writing or Appending to a File

The processes for writing to or appending to a file are the same. The difference lies in the `fopen()` call. When you write to a file, you should use the mode argument `"w"` when you call `fopen()`:

```
$fp = fopen( "test.txt", "w" );
```

All subsequent writing will occur from the start of the file. If the file doesn't already exist, it will be created. If the file already exists, any prior content will be destroyed and replaced by the data you write.

When you append to a file, you should use mode `"a"` in your `fopen()` call:

```
$fp = fopen( "test.txt", "a" );
```

Any subsequent writes to your file are added to the existing content.

Writing to a File with `fwrite()` or `fputs()`

`fwrite()` accepts a file resource and a string. It then writes the string to the file. `fputs()` works in exactly the same way.

```
fwrite( $fp, "hello world" );
fputs( $fp, "hello world" );
```

Writing to files is as straightforward as that. Listing 10.13 uses fwrite() to print to a file. We then append a further string to the same file using fputs().

Listing 10.13 Writing and Appending to a File

```
 1: <html>
 2: <head>
 3: <title>Listing 10.13 Writing and appending to a file</title>
 4: </head>
 5: <body>
```

```
 6: <?php
 7: $filename = "test.txt";
 8: print "Writing to $filename<br>";
 9: $fp = fopen( $filename, "w" ) or die("Couldn't open $filename");
10: fwrite( $fp, "Hello world\n" );
11: fclose( $fp );
12: print "Appending to $filename<br>";
13: $fp = fopen( $filename, "a" ) or die("Couldn't open $filename");
14: fputs( $fp, "And another thing\n" );
15: fclose( $fp );
16: ?>
17: </body>
18: </html>
```

Locking Files with `flock()`

The techniques you have learned for reading and amending files will work fine if you are only presenting your script to a single user. In the real world, however, you would expect many users to access your projects more or less at the same time. Imagine what would happen if two users were to execute a script that writes to one file at the same moment. The file will quickly become corrupt.

PHP 4 provides the `flock()` function to forestall this eventuality. `flock()` will lock a file to warn other process against writing to or reading from a file while the current process is working with it. `flock()` requires a valid file `resource`, and an integer representing the kind of lock you would like to set. PHP 4 provides predefined constants for each of the integers you are likely to need. In Table 10.1 we list three kinds of locks you can apply to a file.

Table 10.1 Integer arguments to the `flock()` function

Constant	*Integer*	*Lock type*	*Description*
LOCK_SH	1	Shared	Allows other processes to read the file but prevents writing (used when reading a file)
LOCK_EX	2	Exclusive	Prevents other processes from either reading from or writing to a file (used when writing to a file)
LOCK_UN	3	Release	Releases a shared or exclusive lock

You should call `flock()` directly after calling `fopen()` and then call it again to release the lock before closing the file.

```
$fp = fopen( "test.txt", "a" ) or die("couldn't open");
flock( $fp, LOCK_EX ); // exclusive lock
// write to the file
flock( $fp, LOCK_UN ); // release the lock
fclose( $fp );
```

TIP

Locking with `flock()` is advisory. Only other scripts that use `flock()` will respect a lock that you set.

Working with Directories

Now that you can test, read, and write to files, turn your attention to directories. PHP provides many functions to work with directories. You will look at how to create, remove, and read them.

Creating Directories with `mkdir()`

`mkdir()` enables you to create a directory. `mkdir()` requires a string representing the path to the directory you want to create and an integer that should be an octal number representing the mode you want to set for the directory. You specify an octal (base 8) number with a leading 0. The mode argument will only have an effect on Unix systems. The mode should consist of three numbers between 0 and 7, representing permissions for the directory owner, group, and everyone, respectively. This function returns `true` if it successfully creates a directory, or `false` if it doesn't. If `mkdir()` fails, this will usually be because the containing directory has permissions that preclude processes with the script's user ID from writing. If you are not comfortable with setting Unix directory permissions, you should find that one of the examples below fits your needs. Unless you really need your directory to be world writable, you should probably use 0755, which allows the world to read your directory but not write to it.

```
mkdir( "testdir", 0777 ); // global read/write/execute permissions
mkdir( "testdir", 0755 ); // world and group: read/execute only
             // owner: read/write/execute
```

Removing a Directory with `rmdir()`

`rmdir()` enables you to remove a directory from the file system, if the process running your script has the right to do so, and if the directory is empty. `rmdir()` requires only a string representing the path to the directory you want to create.

```
rmdir( "testdir" );
```

Opening a Directory for Reading with `opendir()`

Before you can read the contents of a directory, you must first obtain a directory `r`esource. You can do this with the `opendir()` function. `opendir()` requires a string representing the path to the directory you want to open. `opendir()` returns a directory handle unless the directory is not present or readable, in which case it returns `false`.

```
$dh = opendir( "testdir" );
```

Reading the Contents of a Directory with `readdir()`

Just as you use `gets()` to read a line from a file, you can use `readdir()` to read a file or directory name from a directory. `readdir()` requires a directory handle and returns a string containing the item name. If the end of the directory has been reached, `readdir()` returns `false`. Note that `readdir()` returns only the names of its items, rather than full paths. Listing 10.14 shows the contents of a directory.

Listing 10.14 Listing the Contents of a Directory with `readdir()`

```
 1: <html>
 2: <head>
 3: <title>Listing 10.14 Listing the contents
 4: of a directory with readdir()</title>
 5: </head>
 6: <body>
 7: <?php
 8: $dirname = ".";
 9: $dh = opendir( $dirname ) or die("couldn't open directory");
10:
11: while ( ! ( ( $file = readdir( $dh ) ) === false ) ) {
12:   if ( is_dir( "$dirname/$file" ) )
13:     print " ";
14:   print "$file<br>";
15: }
16: closedir( $dh );
17: ?>
18: </body>
19: </html>
```

We open our directory for reading with the opendir() function on line 9 and use a while statement to loop through each of its elements on line 11. We call readdir() as part of the while statement's test expression, assigning its result to the $file variable. Within the body of the while statement, we use the $dirname variable in conjunction with the $file variable to create a full file path, which we can then test on line 12. If the path leads to a directory, we print " " to the browser on line 13. Finally, we print the filename on line 15.

We have used a cautious construction in the test of the `while` statement. Most PHP programmers (myself included) would use something like the following:

```
while ( $file = readdir( $dh ) ) {
   print "$file<BR>\n";
}
```

The value returned by readdir() will be tested. Because any string other than "0" will resolve to true, there should be no problem. Imagine, however, a directory that contains four files, "0", "1", "2", and "3". The output from the preceding code on my system is as follows:

```
.
..
```

When the loop reaches the file named "0", the string returned by readdir() resolves to false, causing the loop to end. The approach in Listing 10.14 uses === to check that the return value returned by readdir() is not *exactly* equivalent to false. 0 only *resolves* to false in the test, so we circumvent the problem.

Summary

In this hour, you learned how to use `include()` to incorporate files into your documents and to execute any PHP code contained in include files. You learned how to use some of PHP's file test functions. You explored functions for reading files by the line, by the character, or in arbitrary chunks. You learned how to write to files, either replacing or appending to existing content. Finally, you learned how to create, remove, and read directories.

Now that we can work with files, we can save and access substantial amounts of data. If we need to look up data from large files, however, our scripts will begin to slow down quite considerably. What we need is some kind of database. In the next hour we will look at PHP's DBA functions, which give us relatively fast access to data on our filesystem.

Q&A

Q. **Will the `include()` statement slow down my scripts?**

A. Because an included file must be opened and parsed by the engine, it will add some overhead. The benefits of reusable code libraries often outweigh the relatively low performance overhead, however.

Q. **Should I always end script execution if a file cannot be opened for writing or reading?**

A. You should always allow for this possibility. If your script absolutely depends on the file you want to work with, you might want to use the `die()` function, writing an informative error message to the browser. In less critical situations, you will still need to allow for the failure, perhaps adding it to a log file. You can read more about logging in Hour 22, "Debugging."

Workshop

Quiz

1. What functions could you use to add library code to the currently running script?
2. What function would you use to find out whether a file is present on your file system?
3. How would you determine the size of a file?
4. What function would you use to open a file for reading or writing?
5. What function would you use to read a line of data from a file?
6. How can you tell when you have reached the end of a file?
7. What function would you use to write a line of data to a file?
8. How would you open a directory for reading?
9. What function would you use to read the name of a directory item after you have opened a directory for reading?

Quiz Answers

1. You can use the `require()` or `include()` statements to incorporate PHP files into the current document. You could also use `include_once()` or `require_once()`.
2. You can test for the existence of a file with the `file_exists()` function.
3. The `filesize()` function returns a file's size in bytes.
4. The `fopen()` function opens a file. It accepts the path to a file and a character representing the mode. It returns a file `resource`.
5. The `fgets()` function reads data up to the buffer size you pass it, the end of the line, or the end of the document, whichever comes first.
6. The `feof()` function returns `true` when the file `resource` it is passed has reached the end of the file.

7. You can write data to a file with the `fputs()` function.
8. The `opendir()` function enables you to open a directory for reading.
9. The `readdir()` function returns the name of a directory item from an opened directory.

Activities

1. Create a form that accepts a user's first and second name. Create a script that saves this data to a file.
2. Create a script that reads the data file you created in activity 1. As well as writing its contents to the browser (adding a `<BR>` tag to each line), print a summary that includes the number of lines in the file and the file's size.

© 2009 Pearson Education, Inc. All rights reserved.
800 East 96th Street Indianapolis, Indiana 46240