IBM

The right way to read files with PHP

Learn when to use fopen, fclose, feof, fgets, fgetss, and fscanf

Level: Intermediate

Roger McCoy (rogermccoy@gmail.com), IT Specialist, Freelance Writer and Consultant

13 Feb 2007

Learn how to use the different file functions of PHP. Review basic file functions, such as `fopen`, `fclose`, and `feof`; learn reading functions, such as `fgets`, `fgetss`, and `fscanf`. And discover functions that process entire files in one or two lines of code.

Let us count the ways

One of the joys of dealing with modern programming languages like PHP is the amount of options available. PHP could easily steal the Perl motto, "There's more than one way to do it," especially when it comes to file processing. But with the plethora of options available, what's the best tool for the job? Of course, the real answer depends on your goal when parsing the file, so it's worth the time to explore all your options.

Traditional fopen methods

The `fopen` methods are probably the most familiar to old-time C and C++ programmers because they're more or less the tools you've had under your belt for years if you've worked with these languages. For any of these methods, you go through the standard process of using `fopen` to open the file, a function to read the data, then `fclose` to close the file, as shown in Listing 1.

Listing 1. Opening a file and reading it with fgets

```
$file_handle = fopen("myfile", "r");
while (!feof($file_handle)) {
   $line = fgets($file_handle);
   echo $line;
}
fclose($file_handle);
```

Although these functions are familiar to most long-time programmers, let me break them down. Effectively, you perform the following steps:

1. Open the file. `$file_handle` stores a reference to the file itself.
2. Check whether you are already at the end of the file.
3. Keep reading the file until you are at the end, printing each line as you read it.
4. Close the file.

With that in mind, I'll review each file function used here.

fopen

The `fopen` function creates the connection to the file. I say "creates the connection" because in addition to opening a file, `fopen` can open a URL:

```
$fh = fopen("http://127.0.0.1/", "r");
```

This line of code creates a connection to the page above and allows you to start reading it much like a local file.

Note: The `"r"` used in `fopen` indicates that the file is open for reading only. Because writing to files is beyond the scope of this article, I'm not going to list all the other options. However, you should change `"r"` to `"rb"` if you're

reading from binary files for cross-platform compatibility. You'll see an example of this later.

feof

The `feof` command detects whether you have already read to the end of the file and returns True or False. The loop in Listing 1 continues until you have reached the end of the file "myfile." Note that `feof` also returns False if you're reading a URL and the socket has timed out because you no longer have data to read.

fclose

Skipping ahead to the end of Listing 1, `fclose` serves the opposite function of `fopen`: It closes the connection to the file or URL. You are no longer able to read from the file or socket after this function.

fgets

Jumping back a few lines in Listing 1, you get to the heart of file processing: actually reading the file. The `fgets` function is your weapon of choice for this first example. It grabs a single line of data from your file and returns it as a string. From there, you can print or otherwise process your data. The example in Listing 1 nicely prints out an entire file.

If you decide to limit the size of the data chunks that you'll deal with, you can add an argument to `fgets` to limit the maximum line length. For example, use this code to limit the line to 80 characters:

```
$string = fgets($file_handle, 81);
```

Hearkening back to the "\0" end-of-string terminator in C, set the length to one number higher than you actually want. Thus, the example above uses 81 when you want 80 characters. Get in the habit of remembering to add that extra character whenever you use the line limit on this function.

fread

The `fgets` function is only one of many file-reading functions available. It is one of the more commonly used functions because line-by-line parsing often makes sense. In fact, several other functions provide similar functionality. However, line-by-line parsing is not always what you want.

This is where `fread` comes in. The `fread` function serves a slightly different purpose from `fgets`: It is intended to read from binary files (that is, files that don't consist primarily of human-readable text). Because the concept of "lines" isn't relevant for binary files (logical data constructs are not generally terminated by newlines), you must always specify the number of bytes that you wish to read in.

```
$fh = fopen("myfile", "rb");
$data = fread($file_handle, 4096);
```

The above reads in 4,096 bytes (4 KB) of data. Note that no matter what number you specify, `fread` will not read more than 8,192 bytes (8 KB).

Assuming that the file is no bigger than 8 KB, the code below should read the entire file into a string.

```
$fh = fopen("myfile", "rb");
$data = fread($fh, filesize("myfile"));
fclose($fh);
```

If the file is longer than this, you will have to use a loop to read the rest in.

Working with binary data

Notice that the examples for this function have used a slightly different argument from `fopen`. When dealing with binary data, always remember to include the `b` option in `fopen`. If you skip this, Microsoft® Windows® systems may not process the file correctly because they will handle newlines differently. This may seem irrelevant if you're dealing with a Linux® system (or some other UNIX® variant), but even if you aren't developing for Windows, this makes for good cross-platform maintainability and is simply a good practice to follow.

fscanf

Coming back to string processing, `fscanf` again follows the traditional C file library functions. If you're unfamiliar with it, `fscanf` reads field data into variables from a file.

```
list ($field1, $field2, $field3) = fscanf($fh, "%s %s %s");
```

The formatting strings used for this function are described in many places, such as PHP.net, so I won't reiterate them here. Suffice it to say that the string formatting is extremely flexible. What is worth noting is that all the fields are placed in the return value of the function. (In C, they would be passed as arguments.)

fgetss

The `fgetss` function breaks away from the traditional file functions and gives you a better idea of the power of PHP. The function acts like `fgets`, but strips away any HTML or PHP tags it finds, leaving only naked text. Take the HTML file shown below.

Listing 2. Sample HTML file

```
<html>
    <head><title>My title</title></head>
    <body>
        <p>If you understand what "Cause there ain't no one for to give you no pain"
            means then you listen to too much of the band America</p>
    </body>
</html>
```

Then filter it through the `fgetss` function.

Listing 3. Using fgetss

```
$file_handle = fopen("myfile", "r");
while (!feof($file_handle)) {
   echo = fgetss($file_handle);
}
fclose($file_handle);
```

Here's your output:

```
    My title

        If you understand what "Cause there ain't no one for to give you no pain"
            means then you listen to too much of the band America
```

The fpassthru function

No matter how you've been reading your file, you can dump the rest of your data to your standard output channel using `fpassthru`.

```
fpassthru($fh);
```

Again, this function prints the data, so you don't need to grab the data in a variable.

Nonlinear file processing: Jumping around

Of course, the above functions only allow you to read a file in order. More complex files might require you to jump back and forth to different parts of the file. This is where `fseek` comes in handy.

```
fseek($fh, 0);
```

The above example jumps back to the beginning of a file. If you don't want to go back quite all the way -- let's say a kilobyte into it -- then you just write:

```
fseek($fh, 1024);
```

From PHP V4.0 on, you have a few other options. For example, if you want to jump ahead 100 bytes from your current position, you can try:

```
fseek($fh, 100, SEEK_CUR);
```

Similarly, you can jump back 100 bytes by using:

```
fseek($fh, -100, SEEK_CUR);
```

If you want to jump back 100 bytes before the end of the file, use `SEEK_END`, instead.

```
fseek($fh, -100, SEEK_END);
```

After you've reached the new position, you can use `fgets`, `fscanf`, or anything else to read the data.

Note: You can't use `fseek` on file handles referring to URLs.

Grabbing an entire file

Now we get to some of PHP's more unique file-processing strengths: dealing with massive chunks of data in a line or two. For example, how might you grab a file and display the entire contents on your Web page? Well, you saw an example using a loop with `fgets`. But how can you make this more straightforward? The process is almost ridiculously easy with `fgetcontents`, which places an entire file within a string.

```
$my_file = file_get_contents("myfilename");
echo $my_file;
```

Although it isn't best practice, you can write this command even more concisely as:

```
echo file_get_contents("myfilename");
```

This article is primarily about dealing with local files, but it's worth noting that you can grab, echo, and parse other Web pages with these functions, as well.

```
echo file_get_contents("http://127.0.0.1/");
```

This command is effectively the same as:

```
$fh = fopen("http://127.0.0.1/", "r");
fpassthru($fh);
```

You must be looking at this and thinking, "That's still way too much effort." The PHP developers agree with you. So you can shorten the above command to:

```
readfile("http://127.0.0.1/");
```

The `readfile` function dumps the entire contents of a file or Web page to the default output buffer. By default, this command prints an error message if it fails. To avoid this behavior (if you want to), try:

```
@readfile("http://127.0.0.1/");
```

Of course, if you actually want to parse your files, the single string that `file_get_contents` returns might be a bit overwhelming. Your first inclination might be to break it up a little bit with the `split()` function.

```
$array = split("\n", file_get_contents("myfile"));
```

But why go through all that trouble when there's a perfectly good function to do it for you? PHP's `file()` function does this in one step: It returns an array of strings broken up by lines.

```
$array = file("myfile");
```

It should be noted that there is a slight difference between the above two examples. While the `split` command drops the newlines, the newlines are still attached to the strings in the array when using the `file` command (as with the `fgets` command).

PHP's power goes far beyond this, though. You can parse entire PHP-style .ini files in a single command using `parse_ini_file`. The `parse_ini_file` command accepts files similar to Listing 4.

Listing 4. A sample .ini file

```
; Comment
[personal information]
name = "King Arthur"
quest = To seek the holy grail
favorite color = Blue

[more stuff]
Samuel Clemens = Mark Twain
Caryn Johnson = Whoopi Goldberg
```

The following commands would dump this file into an array, then print that array:

```
$file_array = parse_ini_file("holy_grail.ini");
print_r $file_array;
```

The following output is the result:

Listing 5. Output

```
Array
(
    [name] => King Arthur
    [quest] => To seek the Holy Grail
    [favorite color] => Blue
    [Samuel Clemens] => Mark Twain
    [Caryn Johnson] => Whoopi Goldberg
)
```

Of course, you might notice that this command merged the sections. This is the default behavior, but you can fix it easily by passing a second argument to `parse_ini_file`: `process_sections`, which is a Boolean variable. Set `process_sections` to True.

```
$file_array = parse_ini_file("holy_grail.ini", true);
print_r $file_array;
```

And you'll get the following output:

Listing 6. Output

```
Array
(
    [personal information] => Array
        (
            [name] => King Arthur
            [quest] => To seek the Holy Grail
            [favorite color] => Blue
        )

    [more stuff] => Array
        (
            [Samuel Clemens] => Mark Twain
            [Caryn Johnson] => Whoopi Goldberg
        )

)
```

PHP placed the data into an easily parsable multidimensional array.

This is just the tip of the iceberg when it comes to PHP file processing. More complex functions like `tidy_parse_file` and `xml_parse` can help you handle HTML and XML documents, respectively. See Resources for details on how these particular functions work. These are well worth looking at if you'll be dealing with those types of files, but instead of considering every possible file type you might run into in detail in this article, here are a few good general rules for dealing with the functions I've described thus far.

Good practice

Never assume that everything in your program will work as planned. For example, what if the file you're looking for has moved? What if the permissions have been altered and you're unable to read the contents? You can check for these things in advance by using `file_exists` and `is_readable`.

Listing 7. Use file_exists and is_readable

```
$filename = "myfile";
if (file_exists($filename) && is_readable ($filename)) {
        $fh = fopen($filename, "r");
        # Processing
        fclose($fh);
}
```

In practice, however, such code is probably overkill. Processing the return value of `fopen` is simpler and more accurate.

```
if ($fh = fopen($filename, "r")) {
        # Processing
        fclose($fh);
}
```

Because `fopen` returns False on failure, this will ensure that file processing happens only if the file opens successfully. Of course, if the file is nonexistent or nonreadable, you can expect a negative return value. This makes this single check a catchall for all the problems you might run into. Alternatively, you might have the program exit or display an error message if the open fails.

As with `fopen`, `file_get_contents`, `file`, and `readfile`, all return False on failure to open or process the file. The `fgets`, `fgetss`, `fread`, `fscanf`, and `fclose` functions also return False on error. Of course, with the exception of `fclose`, you are likely already processing the return values on these. With `fclose`, there is little to do if the file handle does not close properly, so checking the return value for `fclose` is generally unnecessary.

Picking your poison

PHP has no shortage of effective ways for reading and parsing files. Classic functions such as `fread` might serve you best much of the time or you might find yourself drawn more to the simplicity of `readfile` when it's just right for the task. It really depends on what you're trying to accomplish.

If you're processing large amounts of data, `fscanf` will probably prove valuable and more efficient than, say, using `file` followed by a `split` and `sprintf` command. In contrast, if you're simply echoing a large amount of text with little modification, `file`, `file_get_contents`, or `readfile` might make more sense. This would likely be the case if you're using PHP for caching or even to create a makeshift proxy server.

PHP gives you a lot of tools for working with files. Become more familiar with each of them and learn which ones best suit the projects you're working on. You've got a lot of options, so make good use of them and have fun processing your files with PHP.

Share this...

Digg this story
Post to del.icio.us
Slashdot it!

Resources

Learn

- PHP.net is the full command reference for all things PHP.
- Read "PHP by example, Part 1" to discover PHP's simplified method for constructing complex and powerful Web-related programs.
- Learn about the `xml_parse` function, which I didn't cover in this article.
- Learn about the `tidy_parse_file` function, which I didn't cover in this article.
- PHP.net is the resource for PHP developers.
- Check out the "Recommended PHP reading list."
- Browse all the PHP content on developerWorks.
- Expand your PHP skills by checking out IBM developerWorks' PHP project resources.
- To listen to interesting interviews and discussions for software developers, check out developerWorks' podcasts.
- Stay current with developerWorks' technical events and webcasts.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Visit Safari Books Online for a wealth of resources for open source technologies.

Get products and technologies

- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Participate in the developerWorks PHP Developer Forum.

About the author

Roger McCoy is a developer who has worked with many programming languages, including C, Java, JavaScript, Perl, PHP, and Microsoft Visual Basic. He has five years of experience developing PHP applications but is perhaps best known for his work as a technician in the call center industry.