

COPYRIGHT NOTICE

This article is Copyright (c) Al Andersen, <http://www.alandersen.com/>

You are free to use this code in your programs providing you adhere to the most current version of the GNU General Public License (GPL), <http://www.gnu.org/licenses/gpl.html>

DISCLAIMERS AND CAVEATS

This code is provided, as is. I don't guarantee that it will work, and I don't guarantee that it will not destroy data on your hard disk. Use at your own risk! Backup important files!

This code requires that a command line version of PHP is installed on your computer.

This code is for use on your local computer only. **Do not use this program on a server – it is not secure!**

RECURSIVE PHP FUNCTION FOR PARSING DIRECTORIES

A task that I continually encounter is to perform some action on a set of photographs. Such actions might be renaming the photo files, resizing them, or maybe creating duplicates that should be stored on some other area on my hard disk. While many of these actions can be performed in a program like Bibble(tm), <http://www.bibblelabs.com/>, by the time I start such a program, make various dialog selections, and have the program execute my selections, I could have already run a small command line program and completed the task.

This tutorial is not about one of these small command line programs. It is about a small function that sits inside of one of these programs that lets you process a set of actions on an entire directory tree, to include the sub-directories beneath it. If you're not familiar with the term *directory*, it is the equivalent of the graphical *folder* represented in pretty drag-n-drop programs. The term *directory* predates such programs, as do I, and that's why I use it.

To process a group of files within a directory, and all the files within its child sub-directories, you need to use recursion. Recursion is simply a function within a program calling itself. For example, if you have a function called *myself*, you can call *myself* from within *myself*, and you'll keep calling *myself* until there is no further need to call it.

There is a danger when using recursion. You need to ensure that at some point you stop calling *myself*. Otherwise, you put your program into what is known as an infinite loop, a situation where it never stops and may lock up your entire computer. Therefore, good program logic is important when writing recursive functions.

Let's put together a quick recursive function to read a directory:

```

function read_directory($p_pathname)
{
    $d = dir ($p_pathname);
    echo "DIRECTORY: {$d->path}\n";

    while (($file = $d->read()) !== false)
    {
        if (($file != ".") and ($file != ".."))
        {
            $filetype = filetype ("{$d->path}/{$file}");

            if ($filetype == "dir")
            {
                read_directory ("{$d->path}/{$file}");
            }
            else
            {
                echo "\tFILE: {$d->path}/{$file}\n";
            }
        }
    }

    $d->close;
}

```

The function is quite simple. It takes a pathname as a parameter. You need to supply the complete path to the directory you want to process. For example, if your home directory is called *myplace* and you want to process the directory *myphotos* in *myplace* you would need to pass `"/home/myplace/myphotos"` to the `read_directory` function. Or, if you have a sub-directory called *vacation* inside of *myphotos*, you would pass `"/home/myplace/myphotos/vacation"` to the `read_directory` function, like so:

```
read_directory ("/home/myplace/myphotos/vacation");
```

So, how does this function work?

First, it returns an instance of the directory object class.

```
$d = dir ($p_pathname);
```

`$d` is now a class object. You can use it to get information about the directory. One such thing would be the pathname of the directory, which is stored in the variable *path* within the class, e.g., use `$d->path` to access the pathname for this directory.

Now, we need to read all the items within the directory. We do this with a `while` loop.

```
while (($file = $d->read()) !== false)
```

This reads the next item in the directory and puts it into the variable `$file`.

Now we need to figure out what to do with each *file* in our directory.

For starters, there are two special files that we do not want to process. First there is `."`, which is a shorthand for the current directory. We're already processing the current directory so we ignore this item. There is also `.."` which is shorthand for the previous directory. We do not want our recursive function to go outside our current directory! The results could be catastrophic. So, we ignore `.."` to ensure we stay inside our directory.

Remember `$file`? It contains the *file* we read with our while loop above. Technically speaking, a directory contains only files. The problem is that not all files are equal. Some files are other directories. We need to determine which files are directories and which files are not. To do this, we use the `filetype` function.

```
$filetype = filetype ("{$d->path}/{$file}");
```

What we've done is combine our directory pathname `$d->path` and the file name `$file` into a single pathname, `"{$d->path}/{$file}"`.

Some of you familiar with PHP might ask what the curly braces are all about. When using quotes, you can enclose a variable in curly braces to have it parsed first. I use this all the time because otherwise I would need to use the PHP concatenation operator (a period) which can be confusing if your editor uses periods to denote tab indentations. So, I prefer `"{$d->path}/{$file}"` to the usual `$d->path . "/" . $file` syntax. It's just cleaner looking.

OK, so now we have the file type and we can test for directory files by using an `if` statement:

```
if ($filetype == "dir")
```

If we find a directory, then we need to stop what we're doing and go parse it, and this is where recursion comes into play. To parse the new directory we call our `read_directory` function, which we are already inside of, and, we pass to `read_directory` the name of the new directory, like so:

```
if ($filetype == "dir")
{
    read_directory ("{$d->path}/{$file}");
}
```

If `$file` is not a directory, then we do something to that file. We could ignore it, but for now, we'll print on the screen that it's a regular file and show its complete pathname

```
else
{
    echo "\tFILE: {$d->path}/{$file}\n";
}
```

Last of all, we have to make sure that we close out our class object. PHP will free classes automatically, but it's good programming practice to explicitly do so, especially here due to the recursive nature of this function.

```
$d->close;
```

You're probably asking, "How does the recursion work?" Well, the function is simple enough that you should be able to look at it and follow the logic. You come into the function with a directory pathname. You open the directory for the pathname and read the files in that directory. If the file type is anything but a directory, it prints the pathname for that file onto the screen, gets the next file, prints its pathname, and keeps doing that until the last file is found and printed. It then closes the directory and exits the function.

However, if the file type is a directory, it stops everything it is doing, remembers the state of all the variables, and then goes and calls itself with the new directory pathname. It then does everything it needs to do for the new directory, and when the function exits itself, it reverts to the remembered state, continuing where it left off with the previous directory. Conceptually this may be difficult to grasp, but it's pretty simple once you wrap a brain cell or two around it.

Go ahead and run the program on a directory. It should be safe to do so, seeing as the program doesn't make any changes to any files and only prints informational messages on the screen. Try it on a directory with only a few files and a few sub-directories within it.

One thing you will note is that the files are processed *as is*. They are not processed in any kind of order, meaning that all the files in Directory A are not processed before the files in Directory B. What actually happens is that Directory A is processed until Directory B is encountered, at which point everything in Directory B needs to be processed before processing in Directory A can continue, and if Directory C is found within Directory B then all of Directory Cs contents must be processed before Directory B can be completed and so on and so on.

The important things to understand is that recursion can be useful. You can process lots of files and sub-directories within a directory using just a little bit of code. Recursion is also dangerous! You can really hose things up if you do something wrong, so test your recursive programs on small test directories before you use them on valuable data.

One more thing to point out, you can use this function non-recursively, meaning you can use it on just the files within the directory you pass to it and not open any sub-directories. To do this, simply comment out the recursive function call, like so:

```
// read_directory ("{$d->path}/{$file}");
```

In our next tutorial I'll show you how to do something useful with this recursive function.