Custom error handlers

string **set_error_handler** (callback *error_handler* [, int *error_types*])

void **restore_error_handler** (void)

int **error_log** (string *message* [, int *message_type* [, string *destination* [, string *extra_headers*]]])

While *assert()* is a good function to make extensive use of, it only catches errors you were expecting. While that might sound obvious, it is quite crucial - if an error you have not planned for occurs, how are you to find out about it? Never fear - there are two functions available to make your life much easier: *set_error_handler()* and *error_log()*.

Set_error_handler() takes the name of a user callback function as its only parameter, and it servers to notify PHP that if there are any errors, it should call that user function to handle them. The user function needs to accept a minimum of two parameters, but in practice you will likely want to accept four. These are, in order, the error number that occurred, the string version of the error, the file the error occurred in, and the line of the error. Here is an example:

```
<?php
    function on_error($num, $str, $file, $line) {
        print "Encountered error $num in $file, line $line: $str\n";
    }

    set_error_handler("on_error");
    print $foo;
?>
```

On line four we define the general error handler to be the *on_error()* function, then call print $foo which, as $foo does not exist, is an error, and will result in *on_error()* being called. The definition of *on_error()* is as described - it takes four parameters, then prints them out to the screen in a nicely formatted manner.

There is a second parameter to *set_error_handler()* that lets you choose what errors should trigger the error handler, and it works like the error_reporting directive in php.ini. However, it's important to remember that you can only have one active error handler at any time, not one for each level of error. This code should explain it:

```
<?php
    function func_notice($num, $str, $file, $line) {
        print "Encountered notice $num in $file, line $line: $str\n";
    }

    function func_error($num, $str, $file, $line) {
        print "Encountered error $num in $file, line $line: $str\n";
    }

    set_error_handler("func_notice", E_NOTICE);
    set_error_handler("func_error", E_ERROR);

    echo $foo;
?>
```

Note that the error is that $foo isn't set; that should output a notice. On the surface that looks as though we're assigning *func_notice()* to handle E_NOTICE-level messages and also assigning *func_error()* to handle E_ERROR-level messages. However, because we can only have one error handler at any one time, the second call to *set_error_handler()* replaces the first with one that only listens to E_ERROR messages.

The *restore_error_handler()* takes no parameters and returns no meaningful value, but it restores the previous error handler. There is only really one potential slip-up here, and that's when you accidentally call *set_error_handler()* twice with the same function name. If you've done this, calling *restore_error_handler()* won't make any change on the surface. Internally it will be using the previous error handler, but as that happens to be same as the other handler it will appear the same!

It's important to note that *set_error_handler()* does stack up previous error handlers neatly, as this script demonstrates:

```
<?php
    function func_notice($num, $str, $file, $line) {
        print "Encountered notice $num in $file, line $line: $str\n";
    }

    function func_error($num, $str, $file, $line) {
        print "Encountered error $num in $file, line $line: $str\n";
    }

    set_error_handler("func_notice", E_NOTICE);
    set_error_handler("func_notice", E_NOTICE);
    set_error_handler("func_notice", E_NOTICE);

    echo $foo;
    set_error_handler("func_notice", E_NOTICE);
    echo $foo;
    restore_error_handler();
    echo $foo;
    restore_error_handler();
    echo $foo;
    restore_error_handler();
    echo $foo;
    restore_error_handler();
    echo $foo;
?>
```

That will only really make sense once you've seen the output:

```
Encountered notice 8 in C:\home\error.php, line 14: Undefined
variable:  foo
Encountered notice 8 in C:\home\error.php, line 18: Undefined
variable:  foo
Encountered notice 8 in C:\home\error.php, line 22: Undefined
variable:  foo
Encountered notice 8 in C:\home\error.php, line 26: Undefined
variable:  foo
Encountered notice 8 in C:\home\error.php, line 30: Undefined
variable:  foo
PHP Notice:  Undefined variable:  foo in C:\home\error.php on line
34
```

So you can see that we need to call *restore_error_handler()* enough times to fully unwind the stack of error handlers, until eventually the default PHP error handler has control and spits out the usual message.

Author's Note: Please note, this is presented merely for your edification: if I hear of people having to call *restore_error_handler()* more than once because I said it was OK, I'll be most upset! ;)

The last function I want to look at here is *error_log()*, which is a great way to get error data saved to disk (or elsewhere) in just one call. At it's simplest, you can pass *error_log()* just one parameter - an error message - and it will log it for you. To get that, edit your php.ini file and set the error_log directive to a location Apache/PHP can write to. For example, /var/log/php_error would be good for Unix, and c:/windows/php_error.log is good for Windows.

With that done (don't forget to restart Apache if necessary!) we can go ahead and use error_log in its most simple form:

```
<?php
    if (!mysqli_connect("localhost", "baduser", "badpass", "baddb")) {
        error_log("Failed to connect to MySQL!");
    }
?>
```

That will output data to our error log. It should also output actual execution errors into the file - something like Warning: mysqli_connect(): Access denied for user: 'baduser@localhost' (Using password: YES) in C:\home\log.php on line 2. If not, enable log_errors in your php.ini file. Note how PHP automatically inserts line breaks for you after each error.

The next two parameters really work in tandem, so I'll cover them together. Parameter two - oddly - takes an integer to determine where your error should be sent: 0 sends it to the error_log (like the default), 1 sends it by email using the *mail()* function, 3 is unused, and 4 saves it to a file of your choice.

The third parameter qualifies the second in that if you set parameter two to be one (send error by email), parameter three should be the email address of the recipient. Similarly if you set parameter two to be three, parameter three should be the filename to save the error to. There is a slight twist to saving to a custom file, because PHP will not do any of the nice formatting for you like it does in the default error log. For example, it won't insert line breaks for you, and neither will it insert timestamps automatically - you need to insert all that yourself. This works out for the best, as it means you have complete control over your custom error log.

Here's a Windows example - note the \r\n for end of line:

```
<?php
    if (!mysqli_connect("localhost", "baduser", "badpass", "phpdb")) {
        error_log("Failed to connect to MySQL!\r\n", 3, "c:/myerror.txt");
    }
?>
```

Author's Note: If you're running a very new version of PHP (that is, released after PHP 5.0.1) you will have the constant PHP_EOL available to you in your scripts, which is automatically set to either \r, \n, or \r\n depending on what platform the script is running.

http://www.hackingwithphp.com/19/8/10/custom-error-handlers