Programming via PHP Repetition

8.1. The `while` statement

One common problem that appears in PHP is what we should do when we want to do a block of code multiple times. For example, perhaps we have executed an SQL query that has multiple rows in its result, and we want to process each of them, though as we write the script we don't know how many there will be.

PHP has a special construct for executing a block of statements multiple times, called the **`while`** statement. Suppose, for instance, that we want a Web form where the user submits a number, and the Web page displays the result of multiplying that number by each integer from 1 to 100. (Ok, it's rather silly, but play along.... We'll see a more compelling example at the chapter's end.) First, our Web form.

```
<form method="post" action="mult_table.php">
<input type="text" name="multiplicand" />
<input type="submit" value="Compute" />
</form>
```

Now we could write `mult_table.php` by having 100 different **`echo`** statements, but that's rather tedious. The **`while`** statement provides a more compelling approach.

```
<?php import_request_variables("pg", "form_"); ?>
<html>
<head>
<title>Multiplication table</title>
</head>

<body>
<h1>Multiplication table for <?php echo $form_multiplicand; ?></h1>

<p>0: 0
<?php
    $multiplier = 1;
    while($multiplier <= 100) {
        $product = $form_multiplicand * $multiplier;
        echo "<br />$multiplier: $product\n";
        $multiplier = $multiplier + 1;
```

```
    }
?></p>

</body>
</html>
```

A **while** statement looks much like an **if** statement: We have the word **while**, followed by a conditional expression enclosed in parentheses, followed by a block of PHP statements enclosed in braces. Like the **if** statement, you should be careful never to insert a spurious colon between the close parenthesis and opening brace, since PHP will interpret this as a **while** statement with an empty body.

When the computer reaches a **while** statement, it tests whether the condition is true or false, and if it is true it executes the statements in the block — so far, it has behaved just like an **if** statement. But with a **while** statement, after it completes executing the statements in the block, it tests the condition again, and if it is still true, it executes the block again. It repeatedly tests the condition and executes the statement block until finally the condition turns out to be false, whereupon it continues with whatever statements follow the statement block.

In our above example, it continues through the **while** statement block exactly 100 times. Each time through the block, one more product is displayed and `$multiplier`becomes one more that it was before. Eventually, `$multiplier` reaches 101; at that point the **while** statement's condition is no longer true, and so there will not be another run through after this.

Each execution of the statement block is called an **iteration**. Our **while** statement here will always perform exactly 100 iteration; but the number of iterations will vary for other **while** statements. A **while** statement that sometimes has zero iterations is common.

A **while** statement is often called a **loop**, because if we imagine drawing a line through each statement as it is executed, we end up with a loop going repeatedly through the body of the **while** statement.

8.2. A multi-row SQL example

Let's look at a more compelling example, involving our forum database. We already created a page for displaying the information about a user, but even more important would be a page that would list the posts currently in the database.

Of course, there will likely be many posts in the result of our SQL query, though we don't know how many. Our **`while`** statement will have a variable (which we call `$row`) that is the index of the current row being listed; it will go up by one with each iteration, and it will stop once we reach the number of rows in the result.

```
<html>
<head>
<title>Current Posts</title>
</head>

<body>
<h1>Current Posts</h1>

<?php
    $db = mysql_connect("localhost:/export/mysql/mysql.sock");
    mysql_select_db("forum", $db);
    $sql = "SELECT subject, body"
        . " FROM Posts";
    $rows = mysql_query($sql, $db);
    if(!$rows) {
        echo "<p>SQL error: " . mysql_error() . "</p>\n";
    } elseif(mysql_num_rows($rows) == 0) {
        echo "<p>There are not yet any posts.</p>\n";
    } else {
        $row = 0;
        while($row < mysql_num_rows($rows)) {
            $post_subject = mysql_result($rows, $row, 0);
            $post_body = mysql_result($rows, $row, 1);

            echo "<h2>$post_subject</h2>\n";
            echo "<p>$post_body</p>\n";

            $row = $row + 1;
        }
    }
?>
</body>
</html>
```

Notice how this PHP script places a **while** statement inside the body of an **if** statement's **else** clause. PHP allows you to nest such statements within the blocks of other statements. You can even place a **while** statement within another **while** statement; this occurs often enough that it is given a particular name: a **nested loop**. An example where a nested loop would be useful would be displaying a full multiplication table: You'd have one loop that would iterate over each row of the table, and another loop inside it that would display each column inside that row.

8.3. Variable update shortcuts

You'll notice in the above two examples both have variables, called **counters**, that count how many iterations we have completed. (Not all **while** statements have counters as in these examples, but many do.) For each, there is a statement of the form `$x = $x + 1;` to step the counter to its next value; this statement is said to**increment** the counter.

Such statements are common enough that PHP includes a shortcut for saying to increment a variable: You can write instead `$x++;` and PHP will add 1 to it. Thus, we could replace the final statement of the **while** statement's block with `$row++;`.

PHP contains other shortcuts, too. Sometimes, you'll find that you have a counter variable that counts *down* rather than up. For this, you can update the variable using `$x--;`, which **decrements** the variable. It is equivalent to the line `$x = $x - 1;`.

Another fairly common thing is to want a variable to be increased by some distance. For this, you can use '+='; for example, if you want `$x` to increase by 10, you can write `$x += 10;`. Suppose, for example, that for some reason we want to avoid using multiplication in our first example of displaying the multiplication table of a number entered by the user. We can accomplish this through repeated addition.

```
$multiplier = 1;
$product = $form_multiplicand;
while($multiplier <= 100) {
    echo "<br />$multiplier: $product\n";
    $multiplier++;
    $product += $form_multiplicand;
}
```

(In fact, it would be more accurate to multiply each time, since with floating-point arithmetic repeated addition will accumulate rounding errors.)

PHP similarly provides '`--`' and '`*=`' shortcuts for decreasing a variable and multiplying it. None of these shortcuts are essential to using PHP, but they turn out to be handy, and you'll find that other PHP programmers use them whenever possible.

One important distinction that sometimes eludes beginners: The fragments `$x += 2` (with the equals sign) and `$x + 2` (without) mean two very different things. The first, `$x += 2`, says to *change* the value to which `$x` refers, whereas the second, `$x + 2` says to compute the sum of `$x` and `2`, but not to change it. Generally, you won't find '`+=`' in the middle of any statements: They should always be the main core of the statement, whose purpose is to alter the value to which a variable applies. (The same principle applies to '`++`', '`--`', and the other shortcuts described in this section.)

8.4. The `for` statement

PHP provides a different shortcut, too, called the **`for`** statement, which applies for the common situation where you want have a block that you wish to execute for each value of a counter variable. We can again modify our multiplication table fragment to illustrate how this would work.

```
for($multiplier = 1; $multiplier <= 100; $multiplier++) {
    $product = $form_multiplicand * $multiplier;
    echo "<br />$multiplier: $product\n";
}
```

Though a convenient shortcut, the **`for`** statement looks at first ugly and a bit confusing. It consists of a set of parentheses following the word **`for`**, and inside the parentheses will be two semicolons dividing the parentheses into three parts, called respectively the *initializer*, *condition*, and *update*. The initializer is applied first; then the condition is tested; and as long as the condition remains true, the computer will execute the statements in the block and then apply the update.

Any **`for`** statement can be translated into a **`while`** statement that works the same, using the following translation:

`for` statement

```
for(initializer; condition; update) {
        block
}
```

equivalent `while`

```
initializer;
while(condition) {
        block
        update
}
```

With practice, the `for` statement is fairly easy to use. Generally speaking, you would only use it to iterate a counter over several values.

Source: http://www.toves.org/books/php/ch08-loop/index.html