Apress™
Books for Professionals by Professionals

Chapter Three: “Expressions, Operators, and Control Structures”

A Programmer’s Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

info@apress.com

CHAPTER 3

Expressions, Operators, and Control Structures

This chapter will introduce several aspects crucial to any programming language, namely, expressions, operators, and control structures. Knowledge of these topics will prove invaluable for creating large and complex PHP applications, as they will make up much of the code. If you are already familiar with languages such as C and Java, much of this chapter will be a review. If these terms and topics are new for you, comprehension of this chapter will be extremely important for your understanding the later chapters of this book.

Expressions

An *expression* is essentially a phrase representing a particular action in a program. All expressions consists of at least one operand and one or more operators. Before delving into a few examples illustrating the use of expressions, an introduction of operands and operators is in order.

Operands

An *operand* is one of the entities being manipulated in an expression. Valid operands can be of any data type discussed in Chapter 2, "Variables and Data Types." You are probably already familiar with the manipulation and use of operands not only through everyday mathematical calculations, but also through prior programming experience. Some examples of operands follow:

```
$a++; // $a is the operand
$sum = $val1 + val2; // $sum, $val1 and $val2 are operands
```

Operators

An *operator* is a symbol that specifies a particular action in an expression. Many operators may be familiar to you. Regardless, it is important to remember that PHP's automatic type conversion will convert types based on the type of operator

placed between the two operands, which is not always the case in other programming languages.

The precedence and associativity of operators are significant characteristics of a programming language (see "Operator Associativity," later in this chapter, for details). Table 3-1 contains a complete listing of all operators, ordered from highest to lowest precedence. Sections following the table discuss each of these topics in further detail.

Table 3-1. PHP's Operators

OPERATOR	ASSOCIATIVITY	PURPOSE
( )	NA	Precedence ordering
new	NA	Object instantiation
! ~	R	Boolean NOT, bitwise NOT
++ —	R	Autoincrement, autodecrement
@	R	Error concealment
/ * %	L	Division, multiplication, modulus
+ - .	L	Addition, subtraction, concatenation
<< >>	L	Shift left, shift right (bitwise)
< <= > >=	NA	Less than, less than or equal to, greater than, greater than or equal to
== != === <>	NA	Is equal to, is not equal to, identical to, is not equal to
& ^ |	L	Bitwise AND, bitwise XOR, bitwise OR
&& ||	L	Boolean AND, boolean OR
?:	R	Ternary operator
= += *= /= .= %=&= |= ^= <<= >>=	R	Assignment operators
AND XOR OR	L	Boolean AND, boolean XOR, boolean OR

Now that the concepts of operands and operators have been introduced, the following examples of expressions will make much more sense:

```
$a = 5;                  // assign integer value 5 to the variable $a
$a = "5";                // assign string value "5" to the variable $a
$sum = 50 + $some_int;   // assign sum of 50 + $some_int to $sum
$wine = "Zinfandel";     // assign "Zinfandel" to the variable $wine
$inventory++;            // increment the variable $inventory by 1
```

More complex types of expressions that enable the programmer to perform more elaborate calculations are also available. An example follows:

```
$total_cost = $cost + ($cost * 0.06); // cost plus sales tax
```

Operator Precedence

Operator precedence is a characteristic of operators that determines the order in which they will evaluate the operands surrounding them. PHP follows the standard precedence rules used in elementary school math class. Let's consider a few examples:

```
$total_cost = $cost + $cost * 0.06;
```

is the same as writing:

```
$total_cost = $cost + ($cost * 0.06);
```

This is because the multiplication operator has higher precedence than that of the addition operator.

Operator Associativity

The *associativity* characteristic of an operator is a specification of how operations of the same precedence (having the same precedence value as displayed in Table 3-1) are evaluated as they are executed. Associativity can be performed in two directions, left to right and right to left. Left-to-right associativity means that the various operations making up the expression are evaluated from left to right. Consider the following example:

```
$value = 3 * 4 * 5 * 7 * 2;
```

is the same as:

```
$value = ((((3 * 4) * 5) * 7) * 2);
```

resulting in the value 840. This is because the multiplication (*) operator is left-to-right associative. In contrast, right-to-left associativity evaluates operators of the same precedence from right to left:

```
$c = 5;
print $value = $a = $b = $c;
```

is the same as:

```
$c = 5;
$value = ($a = ($b = $c));
```

When this expression is evaluated, variables $value, $a, $b, and $c will all contain the value 5. This is because the assignment operator (=) has right-to-left associativity.

Arithmetic Operators

The arithmetic operators, listed in Table 3-2, perform various mathematical operations and will probably be used frequently in most PHP programs. Fortunately they are easy to use.

Table 3-2. Arithmetic Operators

EXAMPLE	LABEL	OUTCOME
$a + $b	Addition	Sum of $a and $b
$a - $b	Subtraction	Difference of $a and $b
$a * $b	Multiplication	Product of $a and $b
$a / $b	Division	Quotient of $a and $b
$a % $b	Modulus	Remainder of $a / $b

Incidentally, PHP provides a vast assortment of predefined mathematical functions, capable of performing base conversions and calculating logarithms, square roots, geometric values, and more. Check the manual for an updated list of these functions.

Assignment Operators

The *assignment operators* assign a data value to a variable. The simplest form of assignment operator just assigns some value, while others (known as *shortcut assignment operators*) perform some other operation before making the assignment. Table 3-3 lists examples using this type of operator.

Table 3-3. Assignment Operators

EXAMPLE	LABEL	OUTCOME
$a = 5;	Assignment	$a equals 5
$a += 5;	Addition-assignment	$a equals $a plus 5
$a *= 5;	Multiplication-assignment	$a equals $a multiplied by 5
$a /= 5;	Division-assignment	$a equals $a divided by 5
$a .= 5;	Concatenation-assignment	$a equals $a concatenated with 5

Prudent use of assignment operators ultimately result in cleaner, more compact code.

String Operators

PHP's *string operators* (see Table 3-4) provide a convenient way in which to concatenate strings together. There are two such operators, including the concatenation operator (.) and the concatenation assignment operator (.=), discussed in the previous section, "Assignment Operators."

DEFINITION *Concatenate means to combine two or more objects together to form one single entity.*

Table 3-4. String Operators

EXAMPLE	LABEL	OUTCOME
$a = "abc"."def";	Concatenation	$a equals the concatenation of the two strings $a and $b
$a .= "ghijkl";	Concatenation-assignment	$a equals its current value concatenated with "ghijkl".

Here is an example of usage of the string operators:

```
// $a will contain string value "Spaghetti & Meatballs";
$a = "Spaghetti" . "& Meatballs";

// $a will contain value "Spaghetti & Meatballs are delicious.".
$a .= "are delicious";
```

The two concatenation operators are hardly the extent of PHP's string-handling capabilities. Read Chapter 8, "Strings and Regular Expressions," for a complete accounting of this functionality.

Autoincrement and Autodecrement Operators

The *autoincrement* (++) and *autodecrement* (—) operators listed in Table 3-5 present a minor convenience in terms of code clarity, providing shortened means by which to add 1 to or subtract 1 from the current value of a variable.

Table 3-5. PHP's Autoincrement and Autodecrement Operators

EXAMPLE	LABEL	OUTCOME
++$a, $a++	Autoincrement	Increment $a by 1
—$a, $a—	Autodecrement	Decrement $a by 1

Interestingly, these operators can be placed on either side of a variable, the side on which they are placed providing a slightly different effect. Consider the outcomes of the following examples:

```
$inventory = 15;                  // Assign integer value 15 to $inventory
$old_inv = $inventory–;           // FIRST assign $old_inv the value of
                                  // $inventory, THEN decrement $inventory.
$orig_inventory = ++$inventory;   // FIRST increment inventory, then assign
                                  // the newly incremented $inventory value
                                  // to $orig_inventory
```

As you can see, the order in which the autoincrement and autodecrement operators are used can have profound effects on the value of a variable.

Logical Operators

Much like the arithmetic operators, *logical operators* (see Table 3-6) will probably play a major role in many of your PHP applications, providing a way to make decisions based on the values of multiple variables. Logical operators make it possible to direct the flow of a program and are used frequently with control structures such as the if conditional and the while and for loops.

Table 3-6. Logical Operators

EXAMPLE	LABEL	OUTCOME
$a && $b	And	True if both $a and $b are true.
$a AND $b	And	True if both $a and $b are true.
$a || $b	Or	True if either $a or $b are true.
$a OR $b	Or	True if either $a or $b are true.
! $a	Not	True if $a is not true.
NOT $a	Not	True if $a is not true.
$a XOR $b	Exclusive or	True if only $a or only $b is true.

Logical operators are also commonly used to provide details about the outcome of other operations, particularly those that return a value:

```
file_exists("filename.txt") OR print "File does not exist!";
```

One of two outcomes will occur:

- The file filename.txt exists
- The sentence "File does not exist!" will be output.

Equality Operators

Equality operators (see Table 3-7) are used to compare two values, testing for equivalence.

Table 3-7. Equality Operators

EXAMPLE	LABEL	OUTCOME
$a == $b	Is equal to	True if $a and $b are equivalent.
$a != $b	Is not equal to	True if $a is not equal to $b
$a === $b	Is identical to	True if $a and $b are equivalent *and* $a and $b have the same type.

It is a common mistake for even experienced programmers to attempt to test for equality using just one equal sign (for example, $a = $b). Keep in mind that this will result in the assignment of the contents of $b to $a, in effect *not* producing the expected results.

Comparison Operators

Comparison operators (see Table 3-8), like logical operators, provide a method by which to direct program flow through examination of the comparative values of two or more variables.

Table 3-8. Comparison Operators

EXAMPLE	LABEL	OUTCOME
$a < $b	Less than	True if $a is less than $b
$a > $b	Greater than	True if $a is greater than $b
$a <= $b	Less than or equal to	True if $a is less than or equal to $b
$a >= $b	Greater than or equal to	True if $a is greater than or equal to $b
($a == 12) ? 5 : -1	Trinary	If $a equals 12, then the return value is 5. Otherwise, the return value is –1.

Note that the comparison operators should be used solely for comparing numerical values. While you may be tempted to compare strings with these operators, you will most likely not arrive at the expected outcome if you do so. There is a set of predefined functions that compare string values. These functions are discussed in detail in Chapter 8, "Strings and Regular Expressions."

Bitwise Operators

Bitwise operators examine and manipulate integer values on the level of individual bits that make up the integer value (thus the name). To fully understand this concept, you must have at least an introductory knowledge to the binary representation of decimal integers. Table 3-9 presents a few decimal integers and their corresponding binary representations.

Table 3-9. Decimal Integers and Their Binary Representations

DECIMAL INTEGER	BINARY REPRESENTATION
2	10
5	101
10	1010
12	1100
145	10010001
1,452,012	101100010011111101100

The bitwise operators listed in Table 3-10 are variations on some of the logical operators, but can result in a drastically different outcome.

Table 3-10. Bitwise Operators

EXAMPLE	LABEL	OUTCOME
$a & $b	And	And together each bit contained in $a and $b
$a | $b	Or	Or together each bit contained in $a and $b
$a ^ $b	Xor	Exclusive-or together each bit contained in $a and $b
~ $b	Not	Negate each bit in $b
$a << $b	Shift left	$a will receive the value of $b shifted left two bits.
$a >> $b	Shift right	$a will receive the value of $b shifted right two bits.

If you are interested in learning more about binary encoding, bitwise operators, and why they are important, I suggest Randall Hyde's massive online reference, "The Art of Assembly Language Programming," available at: http://webster.cs.ucr.edu/Page_asm/Page_asm.html. It's by far the best resource I've found thus far on the Web.

Control Structures

Control structures provide programmers with the tools to build complex programs capable of evaluating and reacting to the changing values of various inputs throughout the execution of a program. In summary, these structures control the execution of a program.

True/False Evaluation

Control structures generally evaluate expressions in terms of true and false. A particular action will occur based on the outcome of this evaluation. Consider the comparative expression `$a = $b`. This expression will evaluate to true if `$a` in fact is equal to `$b`, and false otherwise. More specifically, the expression will evaluate to the value 1 if it is true, and 0 if it is false. Consider the following:

```
$a = 5;
$b = 5;
print $a == $b;
```

This would result in 1 being displayed. Changing `$a` or `$b` to a value other than 5 would result in 0 being displayed.

if

The if statement is a type of selection statement that evaluates an expression and will (or will not) execute a block of code based on the truth or falsehood of the expression. There are two general forms of the if statement:

```
if (expression) {
    statement block
}
```

and

```
if (expression) {
    statement block
}
else {
   statement block
}
```

As stated in the previous section, "True/False Evaluation," the expression evaluates to either true or false. The execution of the statement block depends on the outcome of this evaluation, where a statement block could be either one or several statements. The following example prints out an appropriate statement after evaluating the string value:

```
if ($cooking_weight < 200) {
    print "This is enough pasta (< 200g) for 1-2 people";
}

else {
    print "That's a lot of pasta. Having a party perhaps?";
}
```

If only one statement is to be executed after the evaluation of the expression, then there is no need to include the bracket enclosures:

```
if ($cooking_weight < 100) print "Are you sure this is enough?";
```

elseif

The elseif statement provides another level of evaluation for the if control structure, adding depth to the number of expressions that can be evaluated:

```
if (expression) {
    statement block
}
elseif (expression) {
          statement block
}
```

> **NOTE** *PHP also allows the alternative representation of the elseif statement, that is, else if. Both result in the same outcome, and the alternative representation is only offered as a matter of convenience. The elseif statement is particularly useful when it is necessary to more specifically evaluate values. Note that an elseif statement will* only *be evaluated if the if and elseif statements before it had all evaluated to false.*

```
if ($cooking_weight < 200) {
     print "This is enough pasta (< 200g) for 1-2 people";
}

elseif ($cooking_weight < 500) {
         print "That's a lot of pasta. Having a party perhaps?";
}

else {
         print "Whoa! Who are you cooking for, a football team?";
}
```

Nested if Statements

The ability to nest, or embed, several if statements within one another provides the ultimate level of control in evaluating expressions. Let's explore this concept by expanding on the cooking weight example in the previous sections. Suppose we wanted to evaluate the cooking weight only if the food in question was pasta:

```
// check $pasta value
if ($food == "pasta") {
    // check $cooking_weight value
    if ($cooking_weight < 200) {
        print "This is enough pasta (< 200g) for 1-2 people";
    }
    elseif ($cooking_weight < 500) {
             print "That's a lot of pasta. Having a party perhaps?";
    }
    else {
            print "Whoa! Who are you cooking for, a football team?";
    }
}
```

As you can see from the preceding code listing, nested if statements provide you with greater control over the flow of your program. As your programs grow in size and complexity, you will find nested control statements an indispensable programming tool.

Multiple Expression Evaluation

To further dictate the flow of control in a program, it is possible to simultaneously evaluate several expressions in a control structure:

```
if ($cooking_weight < 0) {
      print "Invalid cooking weight!";
}

elseif ( ($cooking_weight > 0) && ($cooking_weight < 200) ) {
          print "This is enough pasta (< 200g) for 1-2 people";
}

elseif ( ($cooking_weight > 200) && ($cooking_weight < 500) ) {
         print "That's a lot of pasta. Having a party perhaps?";
}

else {
       print "Whoa! Who are you cooking for, a football team?";
}
```

Multiple expression evaluations enable you to set range restrictions, providing greater control over your code flow while simultaneously reducing otherwise redundant control structure calls, resulting in better code readability.

Alternative Enclosure Bracketing

Control structures are enclosed in a set of brackets to clearly signify the various statements making up the structure. Curly brackets ({ }) were introduced earlier. As a convenience for programmers, an alternative format for enclosing control structures exists, as demonstrated here:

```
if (expression) :
   statement block
else :
   statement block
endif;
```

Therefore the following two structures will produce exactly the same outcome:

```
if ($a == $b) {

   print "Equivalent values!";

}
```

```
if ($a == $b) :

   print "Equivalent values!";

endif;
```

while

The while structure provides a way to repetitively loop through a statement block. The number of times the statement block is executed depends on the total times the expression evaluates to true. The general form of the while loop is:

```
while (expression) :
   statement block
endwhile;
```

Let's consider an example of the computation of n-factorial (n!), where n = 5:

```
$n = 5;
$ncopy = $n;
$factorial = 1; // set initial factorial value
while ($n > 0) :
              $factorial = $n * $factorial;
    $n–; // decrement $n by 1
endwhile;

print "The factorial of $ncopy is $factorial.";
```

resulting in:

```
The factorial of 5 is 120.
```

In the preceding example, $n will be decremented at the conclusion of each loop iteration. We want to be sure that the evaluation expression does not evaluate to true when $n = 0, because this would cause $factorial to be multiplied by 0, surely an unwanted result.

NOTE *In regard to this particular algorithm, the evaluation expression actually could be optimized to be $n > 1, because any number multiplied by 1 will not change. Although this is an extremely small gain in terms of execution time, these factors should always be considered as programs grow in size and complexity.*

do..while

A do..while structure works in much the same way as the while structure presented in the previous section, except that the expression is evaluated at the *end* of each iteration. It is important to note that a do..while loop will always execute at least once, whereas a while loop might not execute at all if the condition is first evaluated before entering the loop.

```
do :
      statement block
while (expression);
```

Let's reconsider the previous n-factorial example, this time using the do..while construct:

```
$n = 5;
$ncopy = $n;
$factorial = 1; // set initial factorial value
do {

$factorial = $n * $factorial;
      $n–; // decrement $n by 1

} while ($n > 0);

print "The factorial of $ncopy is $factorial.";
```

Execution of the preceding example will have the same results as its counterpart in the example accompanying the explanation of the while loop.

> **NOTE** *The do..while loop does not support the alternative syntax form (the colon [:] end control enclosure), allowing only usage of curly brackets as an enclosure.*

for

The for loop is simply an alternative means for specifying the duration of iterative loops. It differs from the while loop only in the fact that the iterative value is updated in the statement itself instead of from somewhere in the statement block. As is the case with the while loop, the looping will continue as long as the condition being evaluated holds true. The general form of the for construct is:

```
for (initialization; condition; increment) {
      statement block
}
```

Three components actually make up the conditional. The *initialization* is considered only once, used to assign the initial value of the loop control variable. The *condition* is considered at the start of every repetition and will determine whether or not the next repetition will occur. Finally, the *increment* determines how the loop control variable changes with each iteration. Use of the term *increment* is perhaps misleading because the variable could be either incremented or decremented in accordance with the programmer's intentions. This example illustrates the basic usage of the for loop:

```
for ($i = 10; $i <= 100; $i+=10) :

      print "\$i = $i <br>";            // escaping backslash to suppress
                                        // conversion of $i variable.
endfor;
```

which results in:

```
$i = 10
$i = 20
$i = 30
$i = 40
$i = 50
$i = 60
```

```
$i = 70
$i = 80
$i = 90
$i = 100
```

Summarizing the example, the conditional variable `$i` is initialized to the value 10. The condition is that the loop will continue until `$i` reaches or surpasses the value 100. Finally, `$i` will be increased by 10 on each iteration. The result is that 10 statements are printed, each denoting the current value of `$i`. It is important to note that an assignment operator is used to increment `$i` by 10. This is not without reason, as the PHP for loop will not accept the alternative method for incrementation, that is, the form `$i = $i + 10`.

Interestingly, the above example can be written in a second format, producing the same results:

```
for ($i = 10; $i <= 100; print "\$i = $i <br>", $i+=10) ;
```

Many novice programmers may be questioning the logic behind having more than one method for implementing looping in a programming language, PHP or another language. The reason for this alternate looping implementation is that quite a few variations of the for loop are available.

One interesting variation is the ability to initialize several variables simultaneously, separating each initialization variable with a comma:

```
for ($x=0,$y=0; $x+$y<10; $x++) :

      $y +=2;                      // increment $y by 2
      print "\$y = $y <BR>";       // print value of $y
      $sum = $x + $y;
      print "\$sum = $sum<BR>";    // print value of $sum

endfor;
```

```
$y = 2
$sum = 2
$y = 4
$sum = 5
$y = 6
$sum = 8
$y = 8
$sum = 11
```

The example will repeatedly print out both the current value of $y and the sum of $x and $y. As you can see, `$sum = 11` is printed, even though this sum surpasses the boundary of the conditional (`$x + $y < 10`). This is because on the entrance of that particular iteration, $y was equal to 6 and $x equal to 2. This fell within the terms of the condition, and $x and $y were incremented, respectively. The sum of 11 was output, but on return to the condition, 11 surpassed the limit of 10, and the for loop was terminated.

It is also possible to omit one of the components of the conditional expression. For example, you may want to pass an initialization variable directly into the for loop, without explicitly setting it to any particular value. You may also want to change the increment variable based on a particular condition in the loop. Therefore, it would make no sense to include these in the for loop. Consider the following example:

```
$x = 5;

for (; ; $x += 2) :

          print " $x ";
          if ($x == 15) :
          break;     // break out of this for loop
           endif;

endfor;
```

which results in the following outcome:

```
5 7 9 11 13 15
```

Although there is no difference in function between the for and while looping structures, the for loop arguably promotes a cleaner code structure. This is because a quick glance in the for statement itself provides the programmer with all of the necessary information regarding the mechanics and duration of the structure. Contrast this with the while statement, where one must take extra time to hunt for iterative updates, a task that could be time consuming as a program grows in size.

foreach

The foreach construct is a variation of the for structure, included in the language as a more convenient means to maneuver through arrays. There are two general forms of the foreach statement, each having its own specific purpose:

```
foreach (array_expression as $value) {
            statement
}

foreach (array_expression as $key => $value) {
             statement
}
```

Let's use the first general format in an expression:

```
$menu = array("pasta", "steak", "potatoes", "fish", "fries");

foreach ($menu as $item) {

             print "$item <BR>";

}
```

resulting in:

```
pasta
steak
potatoes
fish
fries
```

In the above example, two points are worth noting. The first is that the foreach construct will automatically reset the array to its beginning position, something that does not occur using other iterative constructs. Second, there is no need to explicitly increment a counter or otherwise move the array forward; This is automatically accomplished through the foreach construct.

The second general format is used for associative arrays:

```
$wine_inventory = array {
     "merlot" => 15,
     "zinfandel" => 17,
     "sauvignon" => 32
}

foreach ($wine_inventory as $i => $item_count) {
     print "$item_count bottles of $i remaining<BR>";
}
```

resulting in:

```
15 bottles of merlot remaining
17 bottles of zinfandel remaining
32 bottles of sauvignon remaining
```

As this example demonstrates, handling arrays becomes rather simple with the foreach statement. For more information regarding arrays, refer to Chapter 5, "Arrays."

switch

The switch statement functions much like an if statement, testing an expression value against a list of potential matches. It is particularly useful when you need to compare many values, as the switch statement provides clean and compact code. The general format of the switch statement is:

```
switch (expression) {
     case (condition) :
          statement block
     case (condition) :
          statement block
          . . .
     default :
         statement block
}
```

The variable to be evaluated is denoted in the expression part of the switch statement. That variable is then compared with each condition, searching for a match. Should a match be found, the corresponding statement block is executed. Should a match not be found, the optional default statement block will execute.

As you will learn in later chapters, PHP is especially valuable for manipulating user input. Assume that the user is presented with a drop-down list containing several choices, each choice resulting in the execution of a different command contained in a case construct. Use of the switch statement would be very practical for implementing this:

```
$user_input = "recipes"; // assume $user_input is passed in to the script

switch ($user_input) :
     case("search") :
          print "Let's perform a search!";
          break;
     case("dictionary") :
          print "What word would you like to look up?";
          break;
     case("recipes") :
          print "Here is a list of recipes…";
          break;
     default:
          print "Here is the menu…";
          break;
endswitch;
```

As you can see, the switch statement offers a clean and concise way in which to order code. The variable denoted in the switch statement (in this case $user_input) will be evaluated by all subsequent case statements in the switch block. If any of the values denoted in a case statement matches the value contained in the variable being compared, the code contained in that case statement block will be executed. The break statement will then cause the execution of subsequent evaluations and code in the switch construct to be terminated. If none of the cases is applicable, the optional default case statement will be activated. If there is no default case and no cases are applicable, the switch statement will simply be exited, and code execution will continue as necessary below it.

It is important to note that the lack of a break statement (discussed in the next section) in a case will cause all subsequent commands in the switch statement to be executed until either a break statement is found or the end of the switch construct is reached. This result of forgetting a break statement is illustrated in the following listing:

```
$value = 0.4;

switch ($value) :
     case (0.4) :
          print "value is 0.4<br>";
     case (0.6) :
          print "value is 0.6<br>";
          break;
     case (0.3) :
          print "value is 0.3<br>";
          break;
```

```
    default :
        print "You didn't choose a value!";
        break;
endswitch;
```

resulting in the following output:

```
value is 0.4
value is 0.6
```

Lack of the break statement will cause not only the print statement contained in the matching case to be output, but also the print statement contained in the following case. Execution of commands in the switch construct then halts due to the break statement following the second print statement.

> **NOTE** *There are no performance gains to be had in choosing between the switch and if statements. The decision to use one or the other is more or less a matter of convenience for the programmer.*

break

More of a statement than a control structure, break is used to immediately exit out of the while, for, or switch structure in which it is contained. The break statement was already introduced to a certain extent in the preceding section, "switch." However, I'll present one more example to thoroughly introduce the use of the break statement. Let's begin with a review of the rather simple break statement syntax:

```
break n;
```

The optional n proceeding the call to break denotes how many levels of control structures will be terminated should the break statement be executed. For example, if a break statement was nested within two while statements, and the break was preceded by '2', then both while statements would be exited immediately. The default n value is 1, noted either by omitting the n value after the break statement or by explicit inclusion of the value. Interestingly, break does not consider an if statement to be a control statement in the sense that it should be exited in accordance with the depth specified by the n value. Be sure to take this into account when making use of this optional n parameter.

Consider use of the break statement in a foreach loop:

```
$arr = array(14, 12, 128, 34, 5);

$magic_number = 128;

foreach ($arr as $val) :

     if ($val == $magic_number) :
          print "The magic number is in the array!";
          break;
     endif;

     print "val is $val <br>";

endforeach;
```

If the magic number is in fact found in the array `$arr` (in this example, it is), there will be no more need to continue looking for the magic number. The following output would result:

```
val is 14
val is 12
The magic number is in the array!
```

Note that the preceding example is provided merely to illustrate usage of the break statement. A predefined array function exists in `in_array()`, which is capable of searching an array for a given value; `in_array()` is discussed in further detail in Chapter 5, "Arrays."

continue

The final PHP construct that we will examine is continue. Execution of a continue in an iterative loop will bypass the rest of the current loop iteration, instead immediately beginning a new one. The general syntax of continue is:

```
continue n;
```

The optional n acts as the opposite of the n accompanying the break statement, specifying to the end of how many levels of enclosing loops the continue statement should skip.

Let's consider an example that incorporates the continue statement. Suppose we wanted to count prime numbers between 0 and some designated boundary.

For sake of simplicity, assume that we have written a function capable of determining whether or not a number is prime. We'll call that function `is_prime()`:

```
$boundary = 558;

for ($i = 0; $i <= $boundary; $i++) :

   if ( ! is_prime($i)) :
      continue;
   endif;

   $prime_counter++;

endfor;
```

If the number is in fact prime, then the if statement block will be bypassed, and `$prime_counter` will be incremented. Otherwise, the continue statement will be executed, resulting in the jump to the beginning of the loop.

The continue statement is certainly not a necessity, as if statements will accomplish the same result.

> **NOTE** *The use of continue in long and complex algorithms can result in unclear and confusing code. I recommend avoiding use of this construct in these cases.*

Project: Develop an Events Calendar

Putting into practice many of the concepts that have been introduced thus far, I'll conclude this chapter with instructions illustrating how to create a Web-based events calendar. This calendar could store information regarding the latest cooking shows, wine-tasting seminars, or whatever else you deem necessary for your needs. This calendar will make use of many of the concepts you've learned thus far and will introduce you to a few others that will be covered in further detail in later chapters.

A simple file will store the information contained in the calendar. Here are the file's contents:

```
July 21, 2000|8 p.m.|Cooking With Rasmus|PHP creator Rasmus Lerdorf discusses the
wonders of cheese.
July 23, 2000|11 a.m.|Boxed Lunch|Valerie researches the latest ham sandwich
making techniques (documentary)
```

```
July 31, 2000|2:30pm|Progressive Gourmet|Forget the Chardonnay; iced tea is the
sophisticated gourmet's beverage of choice.
August 1, 2000|7 p.m.|Coder's Critique|Famed Food Critic Brian rates NYC's hottest
new Internet cafés.
August 3, 2000|6 p.m.|Australian Algorithms|Matt studies the alligator's diet.
```

Our PHP script shown in Listing 3-1 will produce the output seen in Figure 3-1.

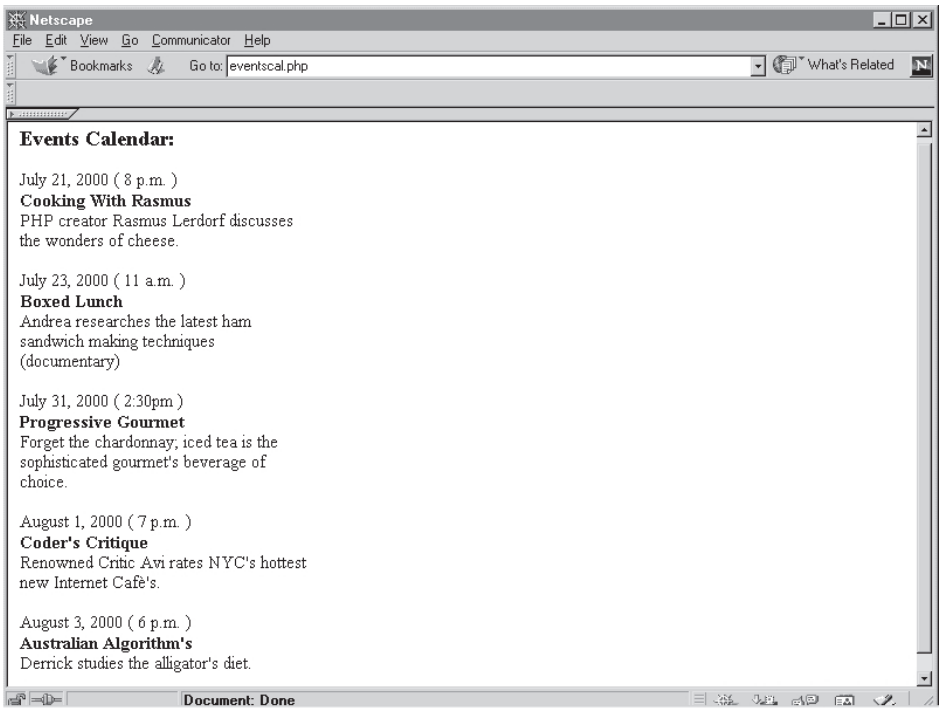

Figure 3-1. The sample events calendar.

Before delving into the code, take a moment to read through the algorithm, which will outline the series of commands executed by the code:

1. Open the file containing the event information.

2. Split each line into four elements: date, time, event title, and event summary.

3. Format and display the event information.

4. Close the file.

Listing 3-1: Script used to display contents of events.txt to browser

```
<?
// application: events calendar
// purpose: read and parse data from a file and format it
// for output to a browser.

// open filehandle entitled '$events' to file 'events.txt'.
$events = fopen("events.txt", "r");

print "<table border = 0 width = 250>";
print "<tr><td valign=top>";

print "<h3>Events Calendar:</h3>";

// while not the end of the file
while (! feof($events)) :

          // read the next line of the events.txt file
          $event = fgets($events, 4096);

          // separate event information in the current
          // line into array elements.

          $event_info = explode("|", $event);

          // Format and output event information
          print "$event_info[0] ( $event_info[1] ) <br>";
          print "<b>$event_info[2]</b> <br>";
          print "$event_info[3] <br> <br>";

     endwhile;
// close the table
print "</td></tr></table>";

fclose ($events);

?>
```

This short example serves as further proof that PHP enables even novice programmers to develop practical applications while investing a minimum of time and learning. Don't worry if you don't understand some of the concepts introduced; they are actually quite simple and will be covered in detail in later

chapters. However, if you just can't wait to learn more about these subjects, jump ahead to Chapter 7, "File I/O and the File System," and Chapter 8, "Strings and Regular Expressions," as much of the unfamiliar syntax is described in those chapters.

What's Next?

This chapter introduced many of the features of the PHP language that you will probably implement in one form or another in almost every script you write: expressions and control structures. Many topics using these features were explained, namely:

- Operators
- Operands
- Operator precedence
- Operator associativity
- Control structures (if, while, do..while, for, foreach, switch, break, continue)

The first three chapters served to introduce you to the core components of the PHP language. The remaining five chapters of this first part of the book will build on these core components, providing you with further information regarding arrays, object-oriented features, file handling, and PHP's string manipulations. This all sets the stage for the second half of the book, serving to highlight PHP's application-building features. So hold on tight and read on!