Apress™
Books for Professionals by Professionals

Chapter Five: “Arrays”

A Programmer’s Introduction to PHP 4.0

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

info@apress.com

CHAPTER 5

Arrays

Chapter 2, "Variables and Data Types," introduced the two types of arrays available for use in your PHP programs, indexed and associative. As you may recall, indexed arrays manipulate elements in accordance with position, while associative arrays manipulate elements in terms of a key/value association. Both offer a powerful and flexible method by which to handle large amounts of data.

This chapter is devoted to the various aspects of PHP's array-manipulation capabilities. By the chapter's conclusion, you will be familiar with single-dimensional and multidimensional arrays, array sorting, array traversal, and various other functions useful in manipulating arrays. It is not in the scope of this book to provide a comprehensive list of all available functions, although this chapter just so happens to cover almost all array functions. For the most up-to-date list, please refer to the PHP home page at `http://www.php.net`.

Creating Arrays

An *array* is essentially a series of objects all bearing the same size and type. Each object in the array is generally known as an *array element*. Creating an array in PHP is easy. You can create an indexed array by placing a set of square brackets ([ ]) after a variable name:

```
$languages[ ] = "Spanish";
// $languages[0] = "Spanish"
```

You can then add further elements to the array, as seen in the following listing. Notice that there is no explicit reference to index positions. Each array allocation is assigned the position at the length of the array plus 1:

```
$languages[ ] = "English"; // $languages[1] = "English"
$languages[ ] = "Gaelic";  // $languages[2] = "Gaelic"
```

You can also explicitly add elements to a particular location by designating the index key:

```
$languages[15] = "Italian";
$languages[22] = "French";
```

You can create associative arrays in much the same way:

```
$languages["Spain"] = "Spanish";
$languages["France"] = "French";
```

There are also three predefined language constructs that you can use to create an array:

- array()
- list()
- range()

Although all achieve the same result, array creation, there are instances in which a given construct may be more suitable than the others. Descriptions and examples of each construct follow.

array()

The array function takes as input zero or more *elements* and returns an array made up of these input elements. Its syntax is:

```
array array ( [element1, element2 …] )
```

The `array()` language construct is perhaps nothing more than a more explicit declaration that an array is being created, used for convenience of the programmer. Here is an example of using `array()` to create an indexed array:

```
$languages = array ("English", "Gaelic", "Spanish");
// $languages[0] = "English", $languages[1] = "Gaelic", $languages[2] = "Spanish"
```

Here is how you would use `array()` to create an associative array:

```
$languages = array ("Spain" => "Spanish",
                    "Ireland" => "Gaelic",
                    "United States" => "English");
// $languages["Spain"] = "Spanish"
// $languages["Ireland"] = "Gaelic"
// $languages["United States"] = "English"
```

Mapping arrays associatively is particularly convenient when using index values just doesn't make sense. In the preceding example it is useful because it

makes sense to associate country names with their language counterparts. Imagine trying to contrive a logical methodology using numbers!

list()

The `list()`language construct is similar to `array()`, though it's used to make simultaneous variable assignments from values extracted from an array in just one operation. Its syntax is:

```
void list (variable1 [, variable2, ...] )
```

It can be particularly useful when extracting information from a database or file. Suppose you wanted to format and output information read from a text file. Each line of the file contains user information, including name, occupation, and favorite color, with each piece of information delimited by a vertical bar (`|`). The typical line would look similar to the following:

```
Nino Sanzi|Professional Golfer|green
```

If you use `list()`, a simple loop could read each line, assign each piece of data to a variable, and format and display the data as needed. Here's how you could use `list()` to make multiple variable assignments:

```
// While the end-of-file hasn't been reached, get next line
while ($line = fgets ($user_file, 4096)) :

     // use split() to separate each piece of data, assign data to $name,
$occupation, and $color
     list ($name, $occupation, $color) = split ( "|", $line);
     // format and output the data
     print "Name: $name <br>";
     print "Occupation: $occupation <br>";
     print "Favorite color: $color <br>";

endwhile;
```

Each line would in turn be read and formatted similar to this:

```
Name: Nino Sanzi
Occupation: Professional Golfer
Favorite Color: green
```

Reviewing the example, `list()` depends on the function `split()` to split each line into three elements. These elements are then assigned to `$name`, `$occupation`, and `$color`, respectively. At that point, it's just a matter of formatting for display to the browser. This is one of the powers of PHP: the ability to easily parse data from text files. This topic is covered in detail in Chapters 7 and 8.

range()

The `range()` language construct provides an easy way to quickly create and fill an array with a specified range of integers, allowing you to specify a range of low and high integer values. An array containing all integer values making up this range is then returned. Its syntax is:

```
array range (int low, int high)
```

You can see the convenience of this construct in the following example:

```
$lottery = range(0,9);
// $lottery = array(0,1,2,3,4,5,6,7,8,9)
```

As you can observe, the range 0 to 9 was specified as the input parameters of `range()`, and the array `$lottery` was subsequently filled with that integer range.

Multidimensional Arrays

As you begin developing more complicated programs, a single-dimensional array may not suffice to store the information that you would like to manipulate. The *multidimensional array* (an array of arrays) offers a much more effective way to store information that requires an extra level of organization. Creating a multidimensional array is easy; simply add an extra set of square brackets to expand the array by one dimension:

```
$chessboard[1] [4] = "King"; // two-dimensional
$capitals["USA"] ["Ohio"] = "Columbus"; // two-dimensional
$streets["USA"] ["Ohio"]["Columbus"] = "Harrison"; // three-dimensional
```

Consider an array that stores information regarding desserts and their preparation details. While this would be rather difficult using a single-dimensional array, a two-dimensional associative array will work just fine:

```
$desserts = array (
                  "Fruit Cup" => array (
                                        "calories" => "low",
                                        "served" => "cold",
                                        "preparation" => "10 minutes"
                                        ),
                  "Brownies" => array (
                                        "calories" => "high",
                                        "served" => "piping hot",
                                        "preparation" => "45 minutes"
                                        )
                  );
```

Once the array has been created, references could be made to each element by indicating the relevant keys:

```
$desserts["Fruit Cup"] ["preparation"] // returns "10 minutes"
$desserts["Brownies"] ["calories"] // returns "high"
```

You can assign elements to a multidimensional array in the same way that you do so with a single-dimensional array:

```
$desserts["Cake"]["calories"] = "too many";
// assigns "too many" to "Cake" property "calories"
```

Although multidimensional arrays introduce another level of complexity to the array structure, creating them is not all that different creating single-dimensional arrays. However, referencing multidimensional arrays in strings requires some special attention. This is the subject of the next section.

Referencing Multidimensional Arrays

You must reference multidimensional arrays in a string slightly differently than you reference other types. You can use the string concatenation operator:

```
print "Brownies are good, but the calorie content is ".
$desserts["Brownies"]["calories"];
```

or you can enclose the multidimensional array in curly brackets ({}):

```
print "Brownies are good, but the calorie content is
{$desserts[Brownies][calories]}";
```

When using this alternative syntax, take note that there are no quotation marks surrounding the array keys. Furthermore, notice that there is no space between the curly brackets and array reference. If you fail to satisfy both of these requisites, an error will occur.

Either way works fine. I suggest choosing one format and sticking with it to eliminate inconsistencies in your code. The flip side to *not* using either of these formatting rules is that the multidimensional array will be interpreted exactly as it is seen in the string, causing what would most certainly be an unexpected outcome.

Locating Array Elements

The ability to easily locate elements in an array is very important. PHP offers a series of functions that allow for the convenient retrieval of both keys and values constituting an array.

in_array()

The `in_array()` function provides a convenient way to quickly determine whether or not an element exists in an array, returning true if it does, and false otherwise. Its syntax is:

```
bool in_array(mixed element, array array)
```

This function is particularly convenient because it eliminates the need to create looping constructs to search through each array element. Consider the following example, which uses in_array() to search for the element "Russian" in the array $languages:

```
$languages = array ("English", "Gaelic", "Spanish");
$exists = in_array("Russian", $languages); // $exists set to false
$exists = in_array("English", $languages); // $exists set to true
```

The `in_array()` function is particularly helpful in a control statement, as the true/false return value can determine the path the conditional construct takes. Here's an example of how you would use `in_array()` to determine the path of a conditional statement:

```
// user input
$language = "French";
$email = "wjgilmore@hotmail.com";
```

```
// if language exists in the array
if (in_array($language, $languages)) :

     // subscribe the user to the newsletter.
     // . Note that subscribe_user() is not a PHP predefined function. I'm just
using it to simulate the process.
     subscribe_user($email, $language);
     print "You are now subscribed to the $language edition of the newsletter.";

// language does not exist in the array
else :
     print "We're sorry, but we don't yet offer a $language edition of the
newsletter".
endif;
```

What happened in this example? Assume that the variables `$language` and `$email` are pieces of data supplied by the user. You want to ensure that their chosen language corresponds to one of those that you offer, and you use `in_array()` to verify this. If it does exist, then the user is subscribed and receives a message stating so. Otherwise, the user is informed that the newsletter is not offered in that particular language. Of course, chances are you are not going to want to force the user to guess in what languages you offer your newsletter. This problem could be eliminated altogether using a drop-down form list, a subject covered in detail in Chapter 10, "Forms." However, for purposes of illustration, this example does the trick nicely.

array_keys()

The `array_keys()` function returns an array containing all of the keys constituting the input array. If the optional `search_element` is included, then only the keys matching that particular element are returned; otherwise, all keys constituting the array are returned. Its syntax is:

```
array array_keys (array array, mixed [search_element])
```

Here's how you could use array_keys() to return the key of a given element:

```
$great_wines = array ("Australia" => "Clarendon Hills 96",
                                  "France" => "Comte Georges de Vogue 97",
                                  "Austria" => "Feiler Artinger 97");

$great_labels = array_keys($great_wines);
```

```
// $great_labels = array ("Australia", "France", "Austria");

$great_labels = array_keys($great_wines, "Clarendon Hills 96");
// $great_labels = array("Australia");
```

Using `array_keys()` is a very easy way to retrieve all of the index values of an array, in the preceding example, the names of the countries where the wines are produced.

array_values()

The `array_values()` function returns an array containing all of the values constituting the input array. Its syntax is:

```
array array_values(array array)
```

Reconsider the previous example, where `array_keys()` was used to retrieve all of the key values. This time array_values() acts to retrieve all of the corresponding key elements:

```
// $great_wines = array ("Australia" = "Clarendon Hills 96",
                         "France = "Comte Georges de Vogue 97",
                         "Austria = "Feiler Artinger 97");

$great_labels = array_values($great_wines);
// $great_labels = array ("Clarendon Hills 96",
                          "Comte Georges de Vogue 97",
                          "Feiler Artinger 97");
```

The `array_keys()` and `array_values()` functions complement each other perfectly, allowing you to retrieve either side of the array as necessary.

Adding and Removing Elements

Thankfully, PHP does not require you to specify the number of elements in an array on its creation. This makes for flexible array manipulation, as there are no worries about surpassing previously designated constraints if an array becomes larger than expected. PHP provides a number of functions for growing an array. Some of these functions are provided as a convenience to programmers wishing to mimic various queue types (FIFO, LIFO, and so on) and stacks, as reflected by their names (push, pop, shift, and unshift). Even if you don't know what queues or stacks are, don't worry; these functions are easy to use.

DEFINITION *A queue is a data structure in which the elements are removed in the same order that they were entered. In contrast, a stack is a data structure in which the elements are removed in the order opposite to that in which they were entered.*

array_push()

The array_push() function appends, or *pushes*, one or more values onto the end of the array. Its syntax is:

```
int array_push(array array, mixed var, [. . . ])
```

The length of the array will increase in direct proportion to the number of values pushed onto the array. This is illustrated in the following example:

```
$languages = array("Spanish", "English", "French");
array_push($languages, "Russian", "German", "Gaelic");
// $languages = array("Spanish", "English", "French",
//                    "Russian", "German", "Gaelic");
```

As is the case with many of PHP's predefined functions, array_push() has a counterpart entitled array_pop(), which acts to pull elements from an array. The main difference between the two is that while array_push() is capable of adding several elements simultaneously, array_pop() can only pull one element off at a time.

array_pop()

The array_pop() function accomplishes a result the exact opposite of that of array_push(), removing, or *popping*, a value from the end of the array. This value is then returned. Its syntax is:

```
mixed array_pop(array array)
```

Each iteration of array_pop() will shorten the length of the array by 1. Consider the following example:

```
$languages = array ("Spanish", "English", "French",
//                  "Russian", "German", "Gaelic");
$a_language = array_pop ($languages); // $a_language = "Gaelic"
$a_language = array_pop ($languages); // $a_language = "German"
// $languages = array ("Spanish", "English", "French", "Russian");
```

The reason for using array_push() and array_pop() is that they provide for a very clean way to both manipulate array elements and control the length without worrying about uninitialized or empty values. They work much more efficiently than attempting to control these factors on your own.

array_shift()

The array_shift() function operates much like array_pop(), except that it removes one element from the beginning (the left side) of the array. All remaining array elements are shifted one unit toward the beginning of the array. Notice that array_shift() has the same syntax as array_pop():

```
mixed array_shift (array array)
```

The important thing to keep in mind is that array_shift() removes the element from the beginning of the array, as shown here:

```
$languages = array("Spanish", "English", "French", "Russian");
$a_language = array_shift($languages); // $a_language = "Spanish";
// $languages = array("English", "French", "Russian");
```

array_unshift()

The array_unshift() function is the counterpart to array_shift(), instead appending values to the beginning of the array and shifting the array to the right. Its syntax is:

```
int array_unshift(array array, mixed var1 [, mixed var2. . .])
```

You can append one or several values simultaneously, the length of the array increasing in direct proportion to the number of values added. An example of appending multiple values follows:

```
$languages = array ("French", "Italian", "Spanish");
array_unshift ($languages, "Russian", "Swahili", "Chinese");
// $languages = array ("Russian", "Swahili", "Chinese",
//                     "French", "Italian", "Spanish");
```

array_pad()

The array_pad() function enables you to quickly expand an array to a precise size, *padding* it with a default value. Its syntax is:

```
array array_pad(array array, int pad_size, mixed pad_value);
```

The input parameter pad_size specifies the new length of the array. The pad_value parameter specifies the default value to which all of the new array positions should be set. Here are several details regarding array_pad() that you should know:

- If pad_size is positive, then the array will be padded to the right; if negative, the array will be padded to the left.
- If the absolute value of pad_size is less than or equal to the length of the array, then no action will be taken.

> **NOTE** *The absolute value of an integer is its value disregarding any negative signs preceding it. For example, the absolute value of both 5 and –5 is 5.*

Here is an array that is padded from the back:

```
$weights = array (1, 3, 5, 10, 15, 25, 50);
$weights = array_pad($weights, 10, 100);
// The result is $weights = array(1, 3, 5, 10, 15, 25, 50, 100, 100, 100)
```

Here is an array that is padded from the front:

```
$weights = array (1, 3, 5, 10, 15, 25, 50);
$weights = array_pad($weights, -10, 100);
// The result is $weights = array(100, 100, 100, 1, 3, 5, 10, 15, 25, 50)
```

This is an incorrect attempt to pad an array:

```
$weights = array (1, 3, 5, 10, 15, 25, 50);
$weights = array_pad ($weights, 3, 100);
// The array $weights remains $weights = array (1, 3, 5, 10, 15, 25, 50)
```

Traversing Arrays

PHP offers a host of functions for traversing the various elements in an array. Used together, they offer a flexible solution for quickly manipulating and outputting array values. You will probably use these functions frequently, as they form the core of almost every array algorithm.

reset()

The function reset() will rewind the internal pointer of the array back to the first element. It also returns the value of the first element. Its syntax follows:

```
mixed reset (array array)
```

Consider the following array:

```
$fruits = array("apple", "orange", "banana");
```

Suppose the pointer in this array is currently set to the element "orange." Executing:

```
$a_fruit = reset ($fruits);
```

will set the pointer back to the beginning of the array, that is, "apple", and return that value if reset() is used as a function. Alternatively, it could be called as simply:

```
reset ($fruits);
```

This will effectively set the pointer back to the initial array element, but will not return a value.

each()

The each() function performs two distinct operations each time it is executed; It returns the key-value pair residing at the current pointer position and advances the pointer to the next element. The syntax is:

```
array each (array array)
```

For convenience, each() actually returns the key and value in a four-element array, the keys of this array being 0, 1, key, and value. The returned key is associated with the keys 0 and key, while the returned value is associated with the keys 1 and value.

This example uses `each()` to return the element found at the current pointer position:

```
// declare array of five elements
$spices = array("parsley", "sage", "rosemary", "thyme", "pepper");
// make sure that array is set at first element
reset($spices);
// create array $a_spice, which will hold four values.
$a_spice = each($spices);
```

Executing the preceding listing, the array `$a_spice` will now contain the following key-value pairs:

- 0 => 0
- 1 => "parsley"
- key => 0
- value => "parsley"

"Parsley" could then be displayed using either of the following statements:

```
print $a_spice[1];
print $a_spice["value"];
```

A common use of the `each()` function is in conjunction with `list()` and a looping construct for cycling through some or all of the elements in an array. Each iteration of `each()` will return either the next key-value pair or false if it has reached the last element in the array. Revisiting the $spices array, you could print all of the values to the screen using the following script:

```
// reset the array pointer
reset ($spices);
// cycle through each key-value pair, printing only the relevant part (the value)
while ( list ($key, $val) = each ($spices) ) :
     print "$val <br>"
endwhile;
```

A more interesting use of `each()`, along with several other functions introduced in this chapter, follows. Listing 5-1 shows how you could use these functions to display a formatted table of countries and languages.

Listing 5-1: Creating an HTML table from array elements

```
// declare associative array of countries and languages
$languages = array ("Country" => "Language",
                    "Spain" => "Spanish",
                    "USA" => "English",
                    "France" => "French",
                    "Russia" => "Russian");

// begin new table
print "<table border=0>";

// move pointer to first element position
reset ($languages);
// extract the first key and element
$hd1 = key ($languages);
$hd2 = $languages[$hd1];

// Print first key and element as table headers
print "<tr><th>$hd1</th><th>$hd2</th></tr>";

// move to next element set
next($languages);

// Print table rows including keys and elements of array
while ( list ($ctry,$lang) = each ($languages)) :
     print "<tr><td>$ctry</td><td>$lang</td></tr>";
endwhile;

// close table
print "</table>";
```

Execution of the preceding code yields the following HTML table:

COUNTRY	LANGUAGE
Spain	Spanish
USA	English
France	French
Russia	Russian

In this example we truly touched on the power of PHP; that is, the ability to mix dynamic code with HTML to produce clean, formatted results of mined information.

end()

The end() function moves the pointer to the last position of the array. Its syntax is:

```
end (array array)
```

next()

The next() function moves the pointer ahead one position before returning the element found at the pointer position. If an advance in the pointer position will move it past the last element of the array, next() will return false. Its syntax is:

```
mixed next (array array)
```

> **NOTE** *A problem with* next() *is that it will also return false for an array element that exists but is empty. If you are interested in merely traversing the array, use* each() *instead.*

prev()

The prev() function operates just like next(), except that it moves the pointer back one position before returning the element found at the pointer position. If the next retreat in pointer position will move it past the first element of the array, prev() will return false. Its syntax is:

```
mixed prev (array array)
```

> **NOTE** *A problem with prev() is that it will also return false for an array element that exists but is empty. If you are interested in merely traversing the array, use* each() *instead.*

array_walk()

The array_walk() function provides an easy way to apply a function to several or all elements in an array. Its syntax is:

```
int array_walk(array array, string func_name, [mixed data])
```

The function, denoted by the input parameter `func_name`, could be used for many purposes, for example, searching for elements having a specific characteristic or actually modifying the values of the array itself. At least two values must be passed into `func_name`: the first is the array value, and the second is the array key. If the optional input parameter data is supplied, then it will be the third value to `func_name`. Here's how you could use `array_walk()` to delete duplicates in an array:

```
function delete_dupes($element) {
     static $last="";
     if ($element == $last)
          unset($element);
     else
          $last=$element;
}

$emails = array("blah@blah.com", "chef@wjgilmore.com", "blah@blah.com");

sort($emails);
reset($emails);
array_walk($emails,"delete_dupes");

// $emails = array("chef@wjgilmore.com", "blah@blah.com");
```

array_reverse()

The `array_reverse()` function provides an easy way to reverse the order of the elements constituting the array. The syntax is:

```
array array_reverse(array array)
```

An example of `array_reverse()` follows:

```
$us_wine_producers = array ("California", "Oregon", "New York", "Washington");
$us_wine_producers = array_reverse ($us_wine_producers);
// $us_wine_producers = array ("Washington", "New York", "Oregon", "California");
```

Performing `array_reverse()` on an associative array will retain the key/value matching, but reverse the array order.

array_flip()

The array_flip() function will exchange ("flip") all key and element values for the array. Its syntax is:

```
array array_flip(array array)
```

Here's how you could use array_flip() to flip all key and element values:

```
$languages = array("Spain" => "Spanish",
                   "France" => "French",
                   "Italy" => "Italian");

$languages = array_flip($languages);

// $languages = array("Spanish" => "Spain",
//                    "French" => "France",
//                    "Italian" => "Italy");
```

Keep in mind that array_flip() only flips the key/value mapping and does *not* reverse the positioning. To reverse the positioning of the elements, use array_reverse().

Array Size

Knowledge of the current size of an array has many applications when coding efficient scripts. Other than using the size for simple referential purposes, perhaps the most common use of the array size is for looping through arrays:

```
$us_wine_producers = array ("Washington", "New York", "Oregon", "California");
for ($i = 0; $i < sizeof ($us_wine_producers); $i++) :
     print "$us_wine_producers[$i]";
endfor;
```

Because the $us_wine_producers array is indexed by integer value, you can use a for loop to iteratively increment a counting variable ($i) and display each element in the array.

sizeof()

The function sizeof() is used to return the number of elements contained in an array. Its syntax is:

```
int sizeof (array array)
```

You will probably use the `sizeof()function` often in your Web applications. A brief example of its usage follows. The previous example is another common usage of the `sizeof()` function.

```
$pasta = array("bowties", "angelhair", "rigatoni");
$pasta_size = sizeof($pasta);
// $pasta_size = 3
```

An alternative, extended form of `sizeof()` is `count()`, next.

count()

The `count()` function performs the same operations as `sizeof()`, returning the number of values contained in an array. Its syntax is:

```
int count (mixed variable)
```

The only difference between `sizeof()` and `count()` is that `count()` provides a bit more information in some situations:

- If the variable exists and is an array, `count()` will return the number of elements contained in the array.
- If the variable exists but is not an array, the value '1' will be returned.
- If the variable does not exist, the value '0' will be returned.

array_count_values()

The `array_count_values()` function is a variation of `sizeof()` and `count()`, instead counting the frequency of the values appearing in the array. Its syntax is:

```
array array_count_values (array array);
```

The returned array will use the values as keys and their corresponding frequencies as the values, as illustrated here:

```
$states = array("OH", "OK", "CA", "PA", "OH", "OH", "PA", "AK");
$state_freq = array_count_values($states);
```

The array $state_freq will now contain the following key/value associations:

```
$state_freq = array("OH" => 3, "OK" => 1, "CA" => 1, "PA" => 2, "AK" => 1);
```

Sorting Arrays

The importance of sorting routines can hardly be understated in the realm of programming and can be seen in action in such online applications as ecommerce sites (sorting categories by alphabetical order), shopping bots (sorting prices), and software search engines (sorting software by number of downloads). PHP offers the nine predefined sorting functions listed in Table 5-1, each sorting an array in a unique fashion.

Table 5-1. Sort Function Summary

FUNCTION	SORT BY	REVERSE SORT?	MAINTAIN KEY/VALUE CORRELATION?
sort	Value	No	No
rsort	Value	Yes	No
asort	Value	No	Yes
arsort	Value	Yes	Yes
ksort	Key	No	Yes
krsort	Key	Yes	Yes
usort	Value	?	No
uasort	Value	?	Yes
uksort	Key	?	Yes

? applies to the user-defined sorting functions, where the order in which the array is sorted depends on the results brought about by the user-defined function.

You are not limited to using predefined criteria for sorting your array information, as three of these functions (`usort(), uasort(), and uksort()`) allow you to introduce array-specific criteria to sort the information any way you please.

sort()

The `sort()` function is the most basic sorting function, sorting array elements from lowest to highest value. Its syntax is:

```
void sort (array array)
```

Nonnumerical elements will be sorted in alphabetical order, according to their ASCII values. This basic example illustrates use of the sort function:

```
// create an array of cities.
$cities = array ("Aprilia", "Nettuno", "Roma", "Venezia", "Anzio");

// sort the cities from lowest to highest value
sort($cities);

// cycle through the array, printing each key and value.
for (reset ($cities); $key = key ($cities); next ($cities)) :
     print "cities[$key] = $cities[$key] <br>";
endfor;
```

Executing the preceding code yields:

```
cities[0] = Anzio
cities[1] = Aprilia
cities[2] = Nettuno
cities[3] = Roma
cities[4] = Venezia
```

As you can see, the `$cities` array has been sorted in alphabetical order. A variation on this algorithm is `asort()`, introduced later in this chapter.

rsort()

The `rsort()` function operates exactly like the `sort()` function, except that it sorts the elements in reverse order. Its syntax is:

```
void rsort (array array)
```

Reconsider the `$cities` array, first introduced in the preceding example:

```
$cities = array ("Aprilia", "Nettuno", "Roma", "Venezia", "Anzio")
rsort($cities);
```

Using `rsort()` to sort the `$cities` array results in the following reordering:

```
cities[0] = Venezia
cities[1] = Roma
cities[2] = Nettuno
cities[3] = Aprilia
cities[4] = Anzio
```

Once again, the `$cities` array is sorted, but this time in reverse alphabetical order. A variation of this function is `arsort()`, described later in this chapter.

asort()

The `asort()` function works much like the previously explained `sort()` function, except that the array indexes maintain their original association with the elements regardless of the new position the element assumes. The function's syntax is:

```
void asort (array array)
```

Revisiting the `$cities` array:

```
$cities = array ("Aprilia", "Nettuno", "Roma", "Venezia", "Anzio");
asort($cities);
```

Use `asort()` to sort the `$cities` array, which yields this new array ordering:

```
cities[4] = Anzio
cities[0] = Aprilia
cities[1] = Nettuno
cities[2] = Roma
cities[3] = Venezia
```

Note the index values and compare them to those in the example accompanying the introduction to `sort()`. This is the differentiating factor between the two functions.

arsort()

The `arsort()` function is a variation of `asort()`, maintaining the original index association but instead sorting the elements in reverse order. Its syntax is:

```
void arsort (array array)
```

Using `arsort()` to sort the `$cities` array:

```
$cities = array ("Aprilia", "Nettuno", "Roma", "Venezia", "Anzio");
arsort($cities);
```

results in the array being sorted in the following order:

```
cities[3] = Venezia
cities[2] = Roma
cities[1] = Nettuno
cities[0] = Aprilia
cities[4] = Anzio
```

Note the index values and compare them to those in the example accompanying the introduction to `rsort()`. This is the differentiating factor between the two functions.

ksort()

The `ksort()` function sorts an array according to its key values, maintaining the original index association. Its syntax is:

```
void ksort (array array)
```

Consider an array slightly different from the original `$cities` array:

```
$wine_producers = array ("America" => "Napa Valley",
                                      "Italy" => "Tuscany",
                                      "Australia" => "Rutherglen",
                                      "France" => "Loire",
                                      "Chile" => "Rapel Valley");
```

Sorting this array using `ksort()`, it would be reordered as follows:

```
"America" => "Napa Valley"
"Australia" => "Rutherglen"
"Chile" => "Rapel Valley"
"France" => "Loire"
"Italy" => "Tuscany"
```

Contrast this to the effects of sorting `$wine_producers` using sort():

```
"America" => "Napa Valley"
"Australia" => "Tuscany"
"Chile" => "Rutherglen"
"France" => "Loire"
"Italy" => "Rapel Valley"
```

Less than optimal results!

krsort()

The `krsort()` function performs the same operations as `ksort()`, except that the key values are sorted in reverse order. Its syntax is:

```
void krsort (array $array)
```

Sorting `$wine_producers` using `krsort()`:

```
$wine_producers = array ("America" => "Napa Valley",
                                      "Italy" => "Tuscany",
                                      "Australia" => "Rutherglen",
                                      "France" => "Loire",
                                      "Chile" => "Rapel Valley");
krsort($wine_producers);
```

yields the following reordering of `$wine_producers`:

```
"Italy" => "Tuscany"
"France" => "Loire"
"Chile" => "Rapel Valley"
"Australia" => "Rutherglen"
"America" => "Napa Valley"
```

For the most part, the sorting functions presented thus far will suit your general sorting requirements. However, occasionally you may need to define your own sorting criteria. This is possible with PHP, through the use of its three predefined functions: `usort()`, `uasort()`, and `uksort()`.

usort()

The sorting function `usort()` provides a way in which to sort an array based on your own predefined criteria. This is possible because `usort()` accepts as an input parameter a function name that is used to determine the sorting order of the data. Its syntax is:

```
void usort(array array, string function_name)
```

The input parameter array is the name of the array that you are interested in sorting, and the parameter `function_name` is the name of the function on which the sorting mechanism will be based. To illustrate just how useful this function can be, assume that you had a long list of Greek vocabulary that you needed to learn for an upcoming history exam. You wanted to sort the words according to length, so that you could study the longer ones first, saving the short ones for when you are more fatigued. You could sort them according to length using `usort()`:

Listing 5-2: Defining sorting criteria with usort()

```
$vocab = array("Socrates","Aristophanes", "Plato", "Aeschylus",
"Thesmophoriazusae");

function compare_length($str1, $str2) {
     // retrieve the lengths of the next two array values
     $length1 = strlen($str1);
     $length2 = strlen($str2);

     // Verify which string is shorter in length.
     if ($length1 == $length2) :
          return 0;
     elseif ($length1 < $length2) :
          return -1;
     else :
          return 1;
     endif;
}
// call usort(), defining the sorting function compare_length()
usort($vocab, "compare_length");

// display the sorted list
while (list ($key, $val) = each ($vocab)) {
     echo "$val<br>";
}
```

In Listing 5-2, the function `compare_length()` defines how the array will be sorted, in this case by comparing the lengths of the passed in elements. Note that you must define two input parameters that represent the next two array elements to be compared. Furthermore, take note that these elements are implicitly passed into the criteria function once `usort()` is called and that all elements are passed through this function automatically.

The functions `uasort()` and `uksort()` are variations of `usort()`, each using the same syntax. The function `uasort()` will sort according to the predefined criteria, except that the key->value correlation will remain the same. The function `uksort()` will also sort according to the predefined criteria, except that the keys will be sorted instead of the values.

Other Useful Functions

This section describes a few functions that are obscure enough to not have a section subtitle, but are nonetheless useful.

array_merge()

The `array_merge()` function merges 1 to N arrays together, appending each to another in the order in which they appear as input parameters. The function's syntax is:

```
array array_merge (array array1, array array2, . . ., array arrayN)
```

The `array_merge()` function provides an easy way to merge several arrays, as shown here:

```
$arr_1 = array ("strawberry", "grape", "lemon");
$arr_2 = array ("banana", "cocoa", "lime");
$arr_3 = array ("peach", "orange");

$arr_4 = array_merge ($arr_2, $arr_1, $arr_3);
// $arr_4 = array("banana", "cocoa", "lime", "strawberry",
//                "grape", "lemon", "peach", "orange");
```

array_slice()

The `array_slice()` function will return a piece of the array, the starting and ending points decided by the offset and optional length input parameters. Its syntax is:

```
array array_slice(array array, int offset, int [length])
```

There are several nuances regarding the input parameters:

- If the offset is positive, the returned slice will start that far away from the beginning of the array.
- If the offset is negative, the returned slice will start that far away from the end of the array.
- If the length is omitted, the returned array will consist of everything from the offset to the end of the array.
- If the length is provided and is positive, the returned slice will have length elements in it.
- If the length is provided and is negative, the returned slice will stop length elements away from the end of the array.

array_splice()

The `array_splice()` function operates somewhat like the function `array_slice`, except that it replaces the designated elements specified by the offset and the optional length input parameters with the elements in the optional array replacement_array. Its syntax is:

```
array_splice(array input_array, int offset, int [length], array
[replacement_array]);
```

There are several factors to keep in mind regarding the input parameters:

- If offset is positive, then the first element to be removed will be offset elements away from the beginning of the array.
- If offset is negative, then the first element to be removed will be offset elements away from the end of the array.
- If length is not provided, all elements starting from offset to the end of the array will be removed.
- If length is provided and is positive, length elements will be removed from the array.

- If length is provided and is negative, elements from offset to length elements away from the end of the array will be removed.

- If `replacement_array` is not specified, then the elements from offset to the optional length will be removed from the input array.

- If `$replacement_array` is specified, it must be enclosed using the `array()` construct, unless `$replacement_array` consists of only one element.

A few examples are in order to fully illustrate the capabilities of this function. Consider the array `$pasta`, below. Each example will manipulate this array in a slightly different manner.

Remove all elements from the fifth element to the end of the array:

```
$pasta = array_splice($pasta, 5);
```

Remove the fifth and sixth elements from the array:

```
$pasta = array_splice($pasta, 5, 2);
```

Replace the third and fourth elements with new elements:

```
$pasta = array_splice($pasta, 5, 2, array("element1", "element2"));
```

Remove all elements from positions 3 to (n 3):

```
$pasta = array_splice($pasta, 5, -3);
```

As illustrated by the preceding examples, `array_splice()` provides a flexible method to remove specific array elements with a minimal amount of code.

shuffle()

The function `shuffle()` will sort the elements of an array in random order. Its syntax is:

```
void shuffle(array array);
```

What's Next?

This chapter introduced arrays and the predefined array-handling functions offered by PHP. In particular, the following concepts were discussed:

- Creation of indexed and associative arrays
- Multidimensional arrays
- Display of multidimensional arrays
- Locating array elements
- Adding and removing elements
- Array size
- Sorting arrays
- Other useful array functions

Arrays provide a very convenient and flexible means for managing information in Web applications. There are several instances in later chapters in which I make use of arrays to improve coding efficiency and clarity.

Chapter 6 continues our survey of PHP's basic functionality, discussing PHP's object-oriented capabilities.