Chapter 4. Working with Arrays

Arrays are collections of related values, such as the data submitted from a form, the names of students in a class, or the populations of a list of cities. In Chapter 2, you learned that a variable is a named container that holds a value. An array is a container that holds multiple values, each distinct from the rest.

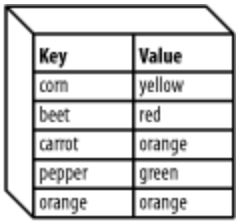
This chapter shows you how to work with arrays. Section 4.1, next, goes over fundamentals such as how to create arrays and manipulate their elements. Frequently, you'll want to do something with each element in an array, such as print it or inspect it for certain conditions. Section 4.2 explains how to do these things with the `foreach( )` and `for( )` constructs. Section 4.3 introduces the `implode( )` and `explode( )` functions, which turn arrays into strings and strings into arrays. Another kind of array modification is sorting, which is discussed in Section 4.4. Last, Section 4.5 explores arrays that themselves contain other arrays.

Chapter 6 shows you how to process form data, which the PHP interpreter automatically puts into an array for you. When you retrieve information from a database as described in Chapter 7, that data is often packaged into an array.

4.1 Array Basics

An array is made up of *elements*. Each element has a *key* and a *value*. An array holding information about the colors of vegetables has vegetable names for keys and colors for values, shown in Figure 4-1.

Figure 4-1. Keys and values



An array can only have one element with a given key. In the vegetable color array, there can't be another element with the key `corn` even if its value is `blue`. However, the same value can appear many times in one array. You can have orange carrots, orange tangerines, and orange oranges.

Any string or number value can be an array element key such as `corn`, `4`, `-36`, or `Salt Baked Squid`. Arrays and other nonscalar[1] values can't be keys, but they can be element values. An element value can be a string, a number, `true`, or `false`; it can also be another array.

[1] *Scalar* describes data that has a single value: a number, a piece of text, true, or false. Complex data types such as arrays, which hold multiple values, are not scalars.

4.1.1 Creating an Array

To create an array, assign a value to a particular array key. Array keys are denoted with square brackets, as shown in Example 4-1.

Example 4-1. Creating arrays

```
// An array called $vegetables with string keys
$vegetables['corn'] = 'yellow';
$vegetables['beet'] = 'red';
$vegetables['carrot'] = 'orange';

// An array called $dinner with numeric keys
$dinner[0] = 'Sweet Corn and Asparagus';
$dinner[1] = 'Lemon Chicken';
$dinner[2] = 'Braised Bamboo Fungus';

// An array called $computers with numeric and string keys
$computers['trs-80'] = 'Radio Shack';
$computers[2600] = 'Atari';
$computers['Adam'] = 'Coleco';
```

The array keys and values in Example 4-1 are strings (such as `corn`, `Braised Bamboo Fungus`, and `Coleco`) and numbers (such as `0`, `1`, and `2600`). They are written just like other strings and numbers in PHP programs: with quotes around the strings but not around the numbers.

You can also create an array using the `array( )` language construct. Example 4-2 creates the same arrays as Example 4-1.

Example 4-2. Creating arrays with array( )

```
$vegetables = array('corn' => 'yellow',
                    'beet' => 'red',
                    'carrot' => 'orange');

$dinner = array(0 => 'Sweet Corn and Asparagus',
                1 => 'Lemon Chicken',
                2 => 'Braised Bamboo Fungus');

$computers = array('trs-80' => 'Radio Shack',
                   2600 => 'Atari',
                   'Adam' => 'Coleco');
```

With `array( )`, you specify a comma-delimited list of key/value pairs. The key and the value are separated by `=>`. The `array( )` syntax is more concise when you are adding more than one element to an array at a time. The square bracket syntax is better when you are adding elements one by one.

4.1.2 Choosing a Good Array Name

Array names follow the same rules as variable names. The first character of an array name must be a letter or number, and the rest of the characters of the name must be letters, numbers, or the underscore. Names for arrays and scalar variables come from the same pool of possible names, so you can't have an array called `$vegetables` and a scalar called `$vegetables` at the same time. If you assign an array value to a scalar or vice versa, then the old value is wiped out and the variable silently becomes the new type. In Example 4-3, `$vegetables` becomes a scalar, and `$fruits` becomes an array.

Example 4-3. Array and scalar collision

```
// This makes $vegetables an array
$vegetables['corn'] = 'yellow';

// This removes any trace of "corn" and "yellow" and makes $vegetables a scalar
$vegetables = 'delicious';

// This makes $fruits a scalar
$fruits = 283;

// This makes $fruits an array and deletes its previous scalar value
$fruits['potassium'] = 'banana';
```

In Example 4-1, the `$vegetables` and `$computers` arrays store a list of relationships. The `$vegetables` array relates vegetables and colors, while the `$computers` array relates computer names and manufacturers. In the `$dinner` array, however, we just care about the names of dishes that are the array values. The array keys are just numbers that distinguish one element from another.

4.1.3 Creating a Numeric Array

PHP provides some shortcuts for working with arrays that have only numbers as keys. If you create an array with `array( )` by specifying only a list of values instead of key/value pairs, the PHP interpreter automatically assigns a numeric key to each value. The keys start at 0 and increase by 1 for each element. Example 4-4 uses this technique to create the `$dinner` array.

Example 4-4. Creating numeric arrays with array( )

```
$dinner = array('Sweet Corn and Asparagus',
                'Lemon Chicken',
                'Braised Bamboo Fungus');
print "I want $dinner[0] and $dinner[1].";
```

Example 4-4 prints:

```
I want Sweet Corn and Asparagus and Lemon Chicken.
```

Internally, the PHP interpreter treats arrays with numeric keys and arrays with string keys (and arrays with a mix of numeric and string keys) identically. Because of the resemblance to features in other programming languages, programmers often refer to arrays with only numeric keys as "numeric," "indexed," or "ordered" arrays, and to string-keyed arrays as "associative" arrays. An *associative array*, in other words, is one whose keys signify something other than the positions of the values within the array.

PHP automatically uses incrementing numbers for array keys when you create an array or add elements to an array with the empty brackets syntax shown in Example 4-5.

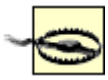
Example 4-5. Adding elements with [ ]

```
// Create $lunch array with two elements
// This sets $lunch[0]
```

```
$lunch[] = 'Dried Mushrooms in Brown Sauce';
// This sets $lunch[1]
$lunch[] = 'Pineapple and Yu Fungus';

// Create $dinner with three elements
$dinner = array('Sweet Corn and Asparagus', 'Lemon Chicken',
                'Braised Bamboo Fungus');
// Add an element to the end of $dinner
// This sets $dinner[3]
$dinner[] = 'Flank Skin with Spiced Flavor';
```

The empty brackets add an element to the array. The element has a numeric key that's one more than the biggest numeric key already in the array. If the array doesn't exist yet, the empty brackets add an element with a key of `0`.



Making the first element have key `0`, not key `1`, is the exact opposite of how normal humans (in contrast to computer programmers) think, so it bears repeating. The first element of an array with numeric keys is element `0`, not element `1`.

4.1.4 Finding the Size of an Array

The `count( )` function tells you the number of elements in an array. Example 4-6 demonstrates `count( )`.

Example 4-6. Finding the size of an array

```
$dinner = array('Sweet Corn and Asparagus',
                'Lemon Chicken',
                'Braised Bamboo Fungus');

$dishes = count($dinner);

print "There are $dishes things for dinner.";
```

Example 4-6 prints:

```
There are 3 things for dinner.
```

When you pass it an empty array (that is, an array with no elements in it), `count( )` returns 0. An empty array also evaluates to `false` in an `if( )` test expression.

4.2 Looping Through Arrays

One of the most common things to do with an array is to consider each element in the array individually and process it somehow. This may involve incorporating it into a row of an HTML table or adding its value to a running total.

The easiest way to iterate through each element of an array is with `foreach( )`. The `foreach( )` construct lets you run a code block once for each element in an array. Example 4-7 uses `foreach( )` to print an HTML table containing each element in an array.

Example 4-7. Looping with foreach( )

```
$meal = array('breakfast' => 'Walnut Bun',
              'lunch' => 'Cashew Nuts and White Mushrooms',
              'snack' => 'Dried Mulberries',
              'dinner' => 'Eggplant with Chili Sauce');
print "<table>\n";
foreach ($meal as $key => $value) {
    print "<tr><td>$key</td><td>$value</td></tr>\n";
}
print '</table>';
```

Example 4-7 prints:

```
<table>
<tr><td>breakfast</td><td>Walnut Bun</td></tr>
<tr><td>lunch</td><td>Cashew Nuts and White Mushrooms</td></tr>
<tr><td>snack</td><td>Dried Mulberries</td></tr>
<tr><td>dinner</td><td>Eggplant with Chili Sauce</td></tr>
</table>
```

For each element in `$meal`, `foreach( )` copies the key of the element into `$key` and the value into `$value`. Then, it runs the code inside the curly braces. In Example 4-7, that code prints `$key` and `$value` with some HTML to make a table row. You can use whatever variable names you want for the key and value inside the code block. If the variable names were in use before the `foreach( )`, though, they're overwritten with values from the array.

When you're using `foreach( )` to print out data in an HTML table, often you want to apply alternating colors or styles to each table row. This is easy to do when you store the alternating color values in a separate array. Then, switch a variable between `0` and `1` each time through the `foreach( )` to print the appropriate color. Example 4-8 alternates between the two color values in its `$row_color` array.

Example 4-8. Alternating table row colors

```
$row_color = array('red','green');
$color_index = 0;
$meal = array('breakfast' => 'Walnut Bun',
              'lunch' => 'Cashew Nuts and White Mushrooms',
              'snack' => 'Dried Mulberries',
              'dinner' => 'Eggplant with Chili Sauce');
print "<table>\n";
foreach ($meal as $key => $value) {
    print '<tr bgcolor="' . $row_color[$color_index] . '">';
    print "<td>$key</td><td>$value</td></tr>\n";
    // This switches $color_index between 0 and 1
    $color_index = 1 - $color_index;
}
print '</table>';
```

Example 4-8 prints:

```
<table>
<tr bgcolor="red"><td>breakfast</td><td>Walnut Bun</td></tr>
<tr bgcolor="green"><td>lunch</td><td>Cashew Nuts and White Mushrooms</td></tr>
<tr bgcolor="red"><td>snack</td><td>Dried Mulberries</td></tr>
<tr bgcolor="green"><td>dinner</td><td>Eggplant with Chili Sauce</td></tr>
</table>
```

Inside the `foreach( )` code block, changing the loop variables like `$key` and `$value` doesn't affect the actual array. If you want to change the array, use the `$key` variable as an index into the array. Example 4-9 uses this technique to double each element in the array.

Example 4-9. Modifying an array with foreach( )

```
$meals = array('Walnut Bun' => 1,
               'Cashew Nuts and White Mushrooms' => 4.95,
               'Dried Mulberries' => 3.00,
               'Eggplant with Chili Sauce' => 6.50);

foreach ($meals as $dish => $price) {
    // $price = $price * 2 does NOT work
    $meals[$dish] = $meals[$dish] * 2;
}

// Iterate over the array again and print the changed values
foreach ($meals as $dish => $price) {
    printf("The new price of %s is \$%.2f.\n",$dish,$price);
}
```

Example 4-9 prints:

```
The new price of Walnut Bun is $2.00.
The new price of Cashew Nuts and White Mushrooms is $9.90.
The new price of Dried Mulberries is $6.00.
The new price of Eggplant with Chili Sauce is $13.00.
```

There's a more concise form of `foreach( )` for use with numeric arrays, shown in Example 4-10.

Example 4-10. Using foreach( ) with numeric arrays

```
$dinner = array('Sweet Corn and Asparagus',
                'Lemon Chicken',
                'Braised Bamboo Fungus');
foreach ($dinner as $dish) {
    print "You can eat: $dish\n";
}
```

Example 4-10 prints:

```
You can eat: Sweet Corn and Asparagus
You can eat: Lemon Chicken
You can eat: Braised Bamboo Fungus
```

With this form of `foreach( )`, just specify one variable name after `as`, and each element value is copied into that variable inside the code block. However, you can't access element keys inside the code block.

To keep track of your position in the array with `foreach( )`, you have to use a separate variable that you increment each time the `foreach( )` code block runs. With `for( )`, you get the position explicitly in your loop variable. The `foreach( )` loop gives you the value of each array element, but the `for( )` loop gives you the position of each array element. There's no loop structure that gives you both at once.

So, if you want to know what element you're on as you're iterating through a numeric array, use `for( )` instead of `foreach( )`. Your `for( )` loop should depend on a loop variable that starts at `0` and continues up to one less than the number of elements in the array. This is shown in Example 4-11.

Example 4-11. Iterating through a numeric array with for( )

```
$dinner = array('Sweet Corn and Asparagus',
                'Lemon Chicken',
                'Braised Bamboo Fungus');
for ($i = 0, $num_dishes = count($dinner); $i < $num_dishes; $i++) {
  print "Dish number $i is $dinner[$i]\n";
}
```

Example 4-11 prints:

```
Dish number 0 is Sweet Corn and Asparagus
Dish number 1 is Lemon Chicken
Dish number 2 is Braised Bamboo Fungus
```

When iterating through an array with `for( )`, you have a running counter available of which array element you're on. Use this counter with the modulus operator to alternate table row colors, as shown in Example 4-12.

Example 4-12. Alternating table row colors with for( )

```
$row_color = array('red','green');
$dinner = array('Sweet Corn and Asparagus',
                'Lemon Chicken',
                'Braised Bamboo Fungus');
print "<table>\n";

for ($i = 0, $num_dishes = count($dinner); $i < $num_dishes; $i++) {
    print '<tr bgcolor="' . $row_color[$i % 2] . '">';
    print "<td>Element $i</td><td>$dinner[$i]</td></tr>\n";
}
print '</table>';
```

Example 4-12 computes the correct table row color with `$i % 2`. This value alternates between 0 and 1 as `$i` alternates between even and odd. There's no need to use a separate variable, such as `$color_index` in Example 4-8, to hold the appropriate row color. Example 4-12 prints:

```
<table>
<tr bgcolor="red"><td>Element 0</td><td>Sweet Corn and Asparagus</td></tr>
<tr bgcolor="green"><td>Element 1</td><td>Lemon Chicken</td></tr>
<tr bgcolor="red"><td>Element 2</td><td>Braised Bamboo Fungus</td></tr>
</table>
```

When you iterate through an array using `foreach( )`, the elements are accessed in the order that they were added to the array. The first element added is accessed first, the second element added is accessed next, and so on. If you have a numeric array whose elements were added in a different order than how their keys would usually be ordered, this could produce unexpected results. Example 4-13 doesn't print out array elements in numeric or alphabetic order.

Example 4-13. Array element order and foreach( )

```
$letters[0] = 'A';
$letters[1] = 'B';
$letters[3] = 'D';
$letters[2] = 'C';

foreach ($letters as $letter) {
    print $letter;
}
```

Example 4-13 prints:

```
ABDC
```

To guarantee that elements are accessed in numerical key order, use `for( )` to iterate through the loop:

```
for ($i = 0, $num_letters = count($letters); $i < $num_letters; $i++) {
    print $letters[$i];
}
```

This prints:

```
ABCD
```

If you're looking for a specific element in an array, you don't need to iterate through the entire array to find it. There are more efficient ways to locate a particular element. To check for an element with a certain key, use `array_key_exists( )`, shown in Example 4-14. This function returns `true` if an element with the provided key exists in the provided array.

Example 4-14. Checking for an element with a particular key

```
$meals = array('Walnut Bun' => 1,
               'Cashew Nuts and White Mushrooms' => 4.95,
               'Dried Mulberries' => 3.00,
               'Eggplant with Chili Sauce' => 6.50,
               'Shrimp Puffs' => 0); // Shrimp Puffs are free!
$books = array("The Eater's Guide to Chinese Characters",
               'How to Cook and Eat in Chinese');

// This is true
if (array_key_exists('Shrimp Puffs',$meals)) {
    print "Yes, we have Shrimp Puffs";
}
// This is false
if (array_key_exists('Steak Sandwich',$meals)) {
    print "We have a Steak Sandwich";
}
// This is true
if (array_key_exists(1, $books)) {
    print "Element 1 is How to Cook in Eat in Chinese";
}
```

To check for an element with a particular value, use `in_array( )`, as shown in Example 4-15.

Example 4-15. Checking for an element with a particular value

```
$meals = array('Walnut Bun' => 1,
               'Cashew Nuts and White Mushrooms' => 4.95,
               'Dried Mulberries' => 3.00,
               'Eggplant with Chili Sauce' => 6.50,
               'Shrimp Puffs' => 0);
$books = array("The Eater's Guide to Chinese Characters",
               'How to Cook and Eat in Chinese');

// This is true: key Dried Mulberries has value 3.00
if (in_array(3, $meals)) {
  print 'There is a $3 item.';
}
// This is true
if (in_array('How to Cook and Eat in Chinese', $books)) {
  print "We have How to Cook and Eat in Chinese";
}
// This is false: in_array( ) is case-sensitive
if (in_array("the eater's guide to chinese characters", $books)) {
  print "We have the Eater's Guide to Chinese Characters.";
}
```

The `in_array( )` function returns `true` if it finds an element with the given value. It is case-sensitive when it compares strings. The `array_search( )` function is similar to `in_array( )`, but if it finds an element, it returns the element key instead of `true`. In Example 4-16, `array_search( )` returns the name of the dish that costs $6.50.

Example 4-16. Finding an element with a particular value

```
$meals = array('Walnut Bun' => 1,
               'Cashew Nuts and White Mushrooms' => 4.95,
               'Dried Mulberries' => 3.00,
               'Eggplant with Chili Sauce' => 6.50,
               'Shrimp Puffs' => 0);

$dish = array_search(6.50, $meals);
if ($dish) {
    print "$dish costs \$6.50";
}
```

Example 4-16 prints:

```
Eggplant with Chili Sauce costs $6.50
```

4.3 Modifying Arrays

You can operate on individual array elements just like regular scalar variables, using arithmetic, logical, and other operators. Example 4-17 shows some operations on array elements.

Example 4-17. Operating on array elements

```
$dishes['Beef Chow Foon'] = 12;
$dishes['Beef Chow Foon']++;
$dishes['Roast Duck'] = 3;

$dishes['total'] = $dishes['Beef Chow Foon'] + $dishes['Roast Duck'];

if ($dishes['total']> 15) {
    print "You ate a lot: ";
}

print 'You ate ' . $dishes['Beef Chow Foon'] . ' dishes of Beef Chow Foon.';
```

Example 4-17 prints:

```
You ate a lot: You ate 13 dishes of Beef Chow Foon.
```

Interpolating array element values in double-quoted strings or here documents is similar to interpolating numbers or strings. The easiest way is to include the array element in the string, but don't put quotes around the element key. This is shown in Example 4-18.

Example 4-18. Interpolating array element values in double-quoted strings

```
$meals['breakfast'] = 'Walnut Bun';
$meals['lunch'] = 'Eggplant with Chili Sauce';
$amounts = array(3, 6);

print "For breakfast, I'd like $meals[breakfast] and for lunch, ";
print "I'd like $meals[lunch]. I want $amounts[0] at breakfast and ";
```

```
print "$amounts[1] at lunch.";
```

Example 4-18 prints:

```
For breakfast, I'd like Walnut Bun and for lunch,
I'd like Eggplant with Chili Sauce. I want 3 at breakfast and
6 at lunch.
```

The interpolation in Example 4-18 works only with array keys that consist exclusively of letters, numbers, and underscores. If you have an array key that has whitespace or other punctuation in it, interpolate it with curly braces, as demonstrated in Example 4-19.

Example 4-19. Interpolating array element values with curly braces

```
$meals['Walnut Bun'] = '$3.95';
$hosts['www.example.com'] = 'web site';

print "A Walnut Bun costs {$meals['Walnut Bun']}.";
print "www.example.com is a {$hosts['www.example.com']}.";
```

Example 4-19 prints:

```
A Walnut Bun costs $3.95.
www.example.com is a web site.
```

In a double-quoted string or here document, an expression inside curly braces is evaluated and then its value is put into the string. In Example 4-19, the expressions used are lone array elements, so the element values are interpolated into the strings.

To remove an element from an array, use `unset( )`:

```
unset($dishes['Roast Duck']);
```

Removing an element with `unset( )` is different than just setting the element value to `0` or the empty string. When you use `unset( )`, the element is no longer there when you iterate through the array or count the number of elements in the array. Using `unset( )` on an array that represents a store's inventory is like saying that the store no longer carries a product. Setting the element's value to `0` or the empty string says that the item is temporarily out of stock.

When you want to print all of the values in an array at once, the quickest way is to use the `implode( )` function. It makes a string by combining all the values in an array and separating them with a string delimiter. Example 4-20 prints a comma-separated list of dim sum choices.

Example 4-20. Making a string from an array with implode( )

```
$dimsum = array('Chicken Bun','Stuffed Duck Web','Turnip Cake');
```

```
$menu = implode(', ', $dimsum);
print $menu;
```

Example 4-20 prints:

```
Chicken Bun, Stuffed Duck Web, Turnip Cake
```

To implode an array with no delimiter, use the empty string as the first argument to `implode( )`:

```
$letters = array('A','B','C','D');
print implode('',$letters);
```

This prints:

```
ABCD
```

Use `implode( )` to simplify printing HTML table rows, as shown in Example 4-21.

Example 4-21. Printing HTML table rows with implode( )

```
$dimsum = array('Chicken Bun','Stuffed Duck Web','Turnip Cake');
print '<tr><td>' . implode('</td><td>',$dimsum) . '</td></tr>';
```

Example 4-21 prints:

```
<tr><td>Chicken Bun</td><td>Stuffed Duck Web</td><td>Turnip Cake</td></tr>
```

The `implode( )` function puts its delimiter between each value, so to make a complete table row, you also have to print the opening tags that go before the first element and the closing tags that go after the last element.

The counterpart to `implode( )` is called `explode( )`. It breaks a string apart into an array. The delimiter argument to `explode( )` is the string it should look for to separate array elements. Example 4-22 demonstrates `explode( )`.

Example 4-22. Turning a string into an array with explode( )

```
$fish = 'Bass, Carp, Pike, Flounder';
$fish_list = explode(', ', $fish);
print "The second fish is $fish_list[1]";
```

Example 4-22 prints:

```
The second fish is Carp
```

4.4 Sorting Arrays

There are several ways to sort arrays. Which function to use depends on how you want to sort your array and what kind of array it is.

The `sort( )` function sorts an array by its element values. It should only be used on numeric arrays, because it resets the keys of the array when it sorts. Example 4-23 shows some arrays before and after sorting.

Example 4-23. Sorting with sort( )

```
$dinner = array('Sweet Corn and Asparagus',
                'Lemon Chicken',
                'Braised Bamboo Fungus');
$meal = array('breakfast' => 'Walnut Bun',
              'lunch' => 'Cashew Nuts and White Mushrooms',
              'snack' => 'Dried Mulberries',
              'dinner' => 'Eggplant with Chili Sauce');

print "Before Sorting:\n";
foreach ($dinner as $key => $value) {
    print " \$dinner: $key $value\n";
}
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}

sort($dinner);
sort($meal);

print "After Sorting:\n";
foreach ($dinner as $key => $value) {
    print " \$dinner: $key $value\n";
}
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}
```

Example 4-23 prints:

```
Before Sorting:
 $dinner: 0 Sweet Corn and Asparagus
 $dinner: 1 Lemon Chicken
 $dinner: 2 Braised Bamboo Fungus
   $meal: breakfast Walnut Bun
   $meal: lunch Cashew Nuts and White Mushrooms
   $meal: snack Dried Mulberries
   $meal: dinner Eggplant with Chili Sauce
After Sorting:
 $dinner: 0 Braised Bamboo Fungus
 $dinner: 1 Lemon Chicken
 $dinner: 2 Sweet Corn and Asparagus
   $meal: 0 Cashew Nuts and White Mushrooms
   $meal: 1 Dried Mulberries
```

```
   $meal: 2 Eggplant with Chili Sauce
   $meal: 3 Walnut Bun
```

Both arrays have been rearranged in ascending order by element value. The first value in `$dinner` is now `Braised Bamboo Fungus`, and the first value in `$meal` is `Cashew Nuts and White Mushrooms`. The keys in `$dinner` haven't changed because it was a numeric array before we sorted it. The keys in `$meal`, however, have been replaced by numbers from `0` to `3`.

To sort an associative array by element value, use `asort( )`. This keeps keys together with their values. Example 4-24 shows the `$meal` array from Example 4-23 sorted with `asort( )`.

Example 4-24. Sorting with asort( )

```
$meal = array('breakfast' => 'Walnut Bun',
              'lunch' => 'Cashew Nuts and White Mushrooms',
              'snack' => 'Dried Mulberries',
              'dinner' => 'Eggplant with Chili Sauce');

print "Before Sorting:\n";
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}

asort($meal);

print "After Sorting:\n";
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}
```

Example 4-24 prints:

```
Before Sorting:
   $meal: breakfast Walnut Bun
   $meal: lunch Cashew Nuts and White Mushrooms
   $meal: snack Dried Mulberries
   $meal: dinner Eggplant with Chili Sauce
After Sorting:
   $meal: lunch Cashew Nuts and White Mushrooms
   $meal: snack Dried Mulberries
   $meal: dinner Eggplant with Chili Sauce
   $meal: breakfast Walnut Bun
```

The values are sorted in the same way with `asort( )` as with `sort( )`, but this time, the keys stick around.

While `sort( )` and `asort( )` sort arrays by element value, you can also sort arrays by key with `ksort( )`. This keeps key/value pairs together, but orders them by key. Example 4-25 shows `$meal` sorted with `ksort( )`.

Example 4-25. Sorting with ksort( )

```
$meal = array('breakfast' => 'Walnut Bun',
```

```
              'lunch' => 'Cashew Nuts and White Mushrooms',
              'snack' => 'Dried Mulberries',
              'dinner' => 'Eggplant with Chili Sauce');

print "Before Sorting:\n";
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}

ksort($meal);

print "After Sorting:\n";
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}
```

Example 4-25 prints:

```
Before Sorting:
   $meal: breakfast Walnut Bun
   $meal: lunch Cashew Nuts and White Mushrooms
   $meal: snack Dried Mulberries
   $meal: dinner Eggplant with Chili Sauce
After Sorting:
   $meal: breakfast Walnut Bun
   $meal: dinner Eggplant with Chili Sauce
   $meal: lunch Cashew Nuts and White Mushrooms
   $meal: snack Dried Mulberries
```

The array is reordered so the keys are now in ascending alphabetical order. Each element is unchanged, so the value that went with each key before the sorting is the same as each key value after the sorting. If you sort a numeric array with `ksort( )`, then the elements are ordered so the keys are in ascending numeric order. This is the same order you start out with when you create a numeric array using `array( )` or `[ ]`.

The array sorting functions `sort( )`, `asort( )`, and `ksort( )` have counterparts that sort in descending order. The reverse-sorting functions are named `rsort( )`, `arsort( )`, and `krsort( )`. They work exactly as `sort( )`, `asort( )`, and `ksort( )` except they sort the arrays so the largest (or alphabetically last) key or value is first in the sorted array, and so subsequent elements are arranged in descending order. Example 4-26 shows `arsort( )` in action.

Example 4-26. Sorting with arsort( )

```
$meal = array('breakfast' => 'Walnut Bun',
              'lunch' => 'Cashew Nuts and White Mushrooms',
              'snack' => 'Dried Mulberries',
              'dinner' => 'Eggplant with Chili Sauce');

print "Before Sorting:\n";
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}
```

```
arsort($meal);

print "After Sorting:\n";
foreach ($meal as $key => $value) {
    print "   \$meal: $key $value\n";
}
```

Example 4-26 prints:

```
Before Sorting:
   $meal: breakfast Walnut Bun
   $meal: lunch Cashew Nuts and White Mushrooms
   $meal: snack Dried Mulberries
   $meal: dinner Eggplant with Chili Sauce
After Sorting:
   $meal: breakfast Walnut Bun
   $meal: dinner Eggplant with Chili Sauce
   $meal: snack Dried Mulberries
   $meal: lunch Cashew Nuts and White Mushrooms
```

4.5 Using Multidimensional Arrays

As mentioned earlier in Section 4.1, the value of an array element can be another array. This is useful when you want to store data that has a more complicated structure than just a key and a single value. A standard key/value pair is fine for matching up a meal name (such as `breakfast` or `lunch`) with a single dish (such as `Walnut Bun` or `Chicken withCashew Nuts`), but what about when each meal consists of more than one dish? Then, element values should be arrays, not strings.

Use the `array( )` construct to create arrays that have more arrays as element values, as shown in Example 4-27.

Example 4-27. Creating multidimensional arrays with array( )

```
$meals = array('breakfast' => array('Walnut Bun','Coffee'),
               'lunch'     => array('Cashew Nuts', 'White Mushrooms'),
               'snack'     => array('Dried Mulberries','Salted Sesame Crab'));

$lunches = array( array('Chicken','Eggplant','Rice'),
                  array('Beef','Scallions','Noodles'),
                  array('Eggplant','Tofu'));

$flavors = array('Japanese' => array('hot' => 'wasabi',
                                     'salty' => 'soy sauce'),
                 'Chinese'  => array('hot' => 'mustard',
                                     'pepper-salty' => 'prickly ash'));
```

Access elements in these arrays of arrays by using more sets of square brackets to identify elements. Each set of square brackets goes one level into the entire array. Example 4-28 demonstrates how to access elements of the arrays defined in Example 4-27.

Example 4-28. Accessing multidimensional array elements

```
print $meals['lunch'][1];           // White Mushrooms
```

```
print $meals['snack'][0];            // Dried Mulberries
print $lunches[0][0];                // Chicken
print $lunches[2][1];                // Tofu
print $flavors['Japanese']['salty']  // soy sauce
print $flavors['Chinese']['hot'];    // mustard
```

Each level of an array is called a *dimension*. Before this section, all the arrays in this chapter are *one-dimensional arrays*. They each have one level of keys. Arrays such as `$meals`, `$lunches`, and `$flavors`, shown in Example 4-28, are called *multidimensional arrays* because they each have more than one dimension.

You can also create or modify multidimensional arrays with the square bracket syntax. Example 4-29 shows some multidimensional array manipulation.

Example 4-29. Manipulating multidimensional arrays

```
$prices['dinner']['Sweet Corn and Asparagus'] = 12.50;
$prices['lunch']['Cashew Nuts and White Mushrooms'] = 4.95;
$prices['dinner']['Braised Bamboo Fungus'] = 8.95;

$prices['dinner']['total'] = $prices['dinner']['Sweet Corn and Asparagus'] +
                             $prices['dinner']['Braised Bamboo Fungus'];

$specials[0][0] = 'Chestnut Bun';
$specials[0][1] = 'Walnut Bun';
$specials[0][2] = 'Peanut Bun';
$specials[1][0] = 'Chestnut Salad';
$specials[1][1] = 'Walnut Salad';
// Leaving out the index adds it to the end of the array
// This creates $specials[1][2]
$specials[1][] = 'Peanut Salad';
```

To iterate through each dimension of a multidimensional array, use nested `foreach( )` or `for( )` loops. Example 4-30 uses `foreach( )` to iterate through a multidimensional associative array.

Example 4-30. Iterating through a multidimensional array with foreach( )

```
$flavors = array('Japanese' => array('hot' => 'wasabi',
                                     'salty' => 'soy sauce'),
                 'Chinese'  => array('hot' => 'mustard',
                                     'pepper-salty' => 'prickly ash'));

// $culture is the key and $culture_flavors is the value (an array)
foreach ($flavors as $culture => $culture_flavors) {

    // $flavor is the key and $example is the value
    foreach ($culture_flavors as $flavor => $example) {
        print "A $culture $flavor flavor is $example.\n";
    }
}
```

Example 4-30 prints:

```
A Japanese hot flavor is wasabi.
A Japanese salty flavor is soy sauce.
A Chinese hot flavor is mustard.
A Chinese pepper-salty flavor is prickly ash.
```

The first `foreach( )` loop in Example 4-30 iterates through the first dimension of `$flavors`. The keys stored in `$culture` are the strings `Japanese` and `Chinese`, and the values stored in `$culture_flavors` are the arrays that are the element values of this dimension. The next `foreach( )` iterates over those element value arrays, copying keys such as `hot` and `salty` into `$flavor` and values such as `wasabi` and `soy sauce` into `$example`. The code block of the second `foreach( )` uses variables from both `foreach( )` statements to print out a complete message.

Just like nested `foreach( )` loops iterate through a multidimensional associative array, nested `for( )` loops iterate through a multidimensional numeric array, as shown in Example 4-31.

Example 4-31. Iterating through a multidimensional array with for( )

```
$specials = array( array('Chestnut Bun', 'Walnut Bun', 'Peanut Bun'),
                   array('Chestnut Salad','Walnut Salad', 'Peanut Salad') );

// $num_specials is 2: the number of elements in the first dimension of $specials
for ($i = 0, $num_specials = count($specials); $i < $num_specials; $i++) {
    // $num_sub is 3: the number of elements in each sub-array
    for ($m = 0, $num_sub = count($specials[$i]); $m < $num_sub; $m++) {
        print "Element [$i][$m] is " . $specials[$i][$m] . "\n";
    }
}
```

Example 4-31 prints:

```
Element [0][0] is Chestnut Bun
Element [0][1] is Walnut Bun
Element [0][2] is Peanut Bun
Element [1][0] is Chestnut Salad
Element [1][1] is Walnut Salad
Element [1][2] is Peanut Salad
```

In Example 4-31, the outer `for( )` loop iterates over the two elements of `$specials`. The inner `for( )` loop iterates over each element of the subarrays that hold the different strings. In the `print` statement, `$i` is the index in the first dimension (the elements of `$specials`), and `$m` is the index in the second dimension (the subarray).

To interpolate a value from a multidimensional array into a double-quoted string or here document, use the curly brace syntax from Example 4-19. Example 4-32 uses curly braces for interpolation to produce the same output as Example 4-31. In fact, the only different line in Example 4-32 is the `print` statement.

Example 4-32. Multidimensional array element value interpolation

```
$specials = array( array('Chestnut Bun', 'Walnut Bun', 'Peanut Bun'),
                   array('Chestnut Salad','Walnut Salad', 'Peanut Salad') );
```

```
// $num_specials is 2: the number of elements in the first dimension of $specials
for ($i = 0, $num_specials = count($specials); $i < $num_specials; $i++) {
    // $num_sub is 3: the number of elements in each sub-array
    for ($m = 0, $num_sub = count($specials[$i]); $m < $num_sub; $m++) {
        print "Element [$i][$m] is {$specials[$i][$m]}\n";
    }
}
```

4.6 Chapter Summary

Chapter 4 covers:

- Understanding the components of an array: elements, keys, and values.
- Defining an array in your programs two ways: with `array( )` and with square brackets.
- Understanding the shortcuts PHP provides for arrays with numeric keys.
- Counting the number of elements in an array.
- Visiting each element of an array with `foreach( )`.
- Alternating table row colors with `foreach( )` and an array of color values.
- Modifying array element values inside a `foreach( )` code block.
- Visiting each element of a numeric array with `for( )`.
- Alternating table row colors with `for( )` and the modulus operator (`%`).
- Understanding the order in which `foreach( )` and `for( )` visit array elements.
- Checking for an array element with a particular key.
- Checking for an array element with a particular value.
- Interpolating array element values in strings.
- Removing an element from an array.
- Generating a string from an array with `implode( )`.
- Generating an array from a string with `explode( )`.
- Sorting an array with `sort( )`, `asort( )`, or `ksort( )`.
- Sorting an array in reverse.
- Defining a multidimensional array.
- Accessing individual elements of a multidimensional array.
- Visiting each element in a multidimensional array with `foreach( )` or `for( )`.
- Interpolating multidimensional array elements in a string.

4.7 Exercises

1. According to the U.S. Census Bureau, the 10 largest American cities (by population) in 2000 were as follows:
    - New York, NY (8,008,278 people)
    - Los Angeles, CA (3,694,820)
    - Chicago, IL (2,896,016)
    - Houston, TX (1,953,631)
    - Philadelphia, PA (1,517,550)
    - Phoenix, AZ (1,321,045)
    - San Diego, CA (1,223,400)

- Dallas, TX (1,188,580)
- San Antonio, TX (1,144,646)
- Detroit, MI (951,270)

Define an array (or arrays) that holds this information about locations and population. Print a table of locations and population information that includes the total population in all 10 cities.

2. Modify your solution to the previous exercise so that the rows in result table are ordered by population. Then modify your solution so that the rows are ordered by city name.
3. Modify your solution to the first exercise so that the table also contains rows that hold state population totals for each state represented in the list of cities.
4. For each of the following kinds of information, state how you would store it in an array and then give sample code that creates such an array with a few elements. For example, for the first item, you might say, "An associative array whose key is the student's name and whose value is an associative array of grade and ID number," as in the following:

```
5.  $students = array('James D. McCawley' => array('grade' => 'A+','id' => 271231),
                      'Buwei Yang Chao' => array('grade' => 'A', 'id' => 818211));
```

a. The grades and ID numbers of students in a class.
b. How many of each item in a store inventory is in stock.
c. School lunches for a week — the different parts of each meal (entree, side dish, drink, etc.) and the cost for each day.
d. The names of people in your family.
e. The names, ages, and relationship to you of people in your family.