CHAPTER 10

Working with Variables

Variables are an important part of any programming language, and that goes for PHP too. Variables are blocks of memory associated with a name and a data type, and variables contain data to be used in calculations, program flow, presentation, and so on.

PHP is a loosely typed language where variables can be used without declarations and where they can change type from line to line, in some cases without losing the content. This makes programming much easier than in more strictly typed languages, but it can also make it more difficult to debug the code.

All variable names in PHP start with a dollar ($) sign. This makes it easy for the scripting engine, as well as the reader, to identify variables anywhere in the code, including when they are embedded in strings. Also, using the $ sign allows the developer to use variable names that would otherwise be reserved by the engine for function names and language constructs. This means writing code where function names are used as variable names, such as `$strlen = strlen("This is a test");`, is allowed.

The first character after the $ sign in a variable name must be a letter or an underscore (_). The remaining characters can be letters, numbers, and underscores, and there is no limit on the length of a variable name (but it makes sense to keep them short and meaningful to ensure the readability of the code). Using short variable names means less typing when writing the code, and using longer names means more descriptive names. Valid letters are any of the characters a–z, the characters A–Z, and any ASCII character from 127 to 255. This makes it possible to use international characters when naming variables. `$LøbeNummer` is a valid variable name but most likely readable only to Danish developers. We prefer to keep variable and function names as well as all comments in English like all the language constructs and built-in functions.

It is also important to note that although function names are case-insensitive in PHP, this is not the case for variables. `$MyVar` and `$myvar` are two different variables in PHP, and this is often the cause of scripting warnings. If PHP is configured to hide errors and warnings, it will be difficult to catch programming errors caused by the misspelling of variables as well as other mistakes. It is recommended to configure PHP (on the development system) to display all errors and warnings; you can do this by defining these two values in `php.ini`:

```
error_reporting  =  E_ALL
display_errors = On
```

■ **Note** On a production site it is good practice to hide most or all errors and warnings from the user, but during development it makes sense to display as much information as possible so you can correct errors.

10-1. Using Variable Types

PHP implements a number of variable types. Any variable can be assigned a value of any of these types or the special NULL value. The special NULL value is not case-sensitive, so NULL and null are the same value. When a variable is assigned the NULL value, it does not have a type, and it is considered to be empty. Table 10-1 lists all types that can be used in PHP.

Table 10-1. *PHP Data Types*

Type	Description
Boolean	Possible values are `True` and `False`.
Float	Floating-point values.
Integer	Integer values.
String	Any series of ASCII characters 0–255. PHP strings are binary safe.
Array	An indexed list of other values. All data types are allowed as values.
Object	A class instance.
Resource	A handle to an internal data structure. This can be a database connection or a result set.

Variables of the types boolean, float, and integer use a fixed amount of memory, and the remaining types use memory as needed; if additional memory is needed, the engine automatically allocates it.

The internal representation of a string value has two parts—the string data and the length. This causes the function `strlen()` to be very efficient, as it will return the stored length value without having to count the number of characters in the string. It also allows a string to contain any of the 256 available ASCII values, so you can use a string to store the content of any file or other form of binary data.

PHP's array implementation is an indexed list of values. The index is often called the *key*, and it can be either an integer or a string value. If boolean or float values are used as keys, they are converted to integers before the value is added or updated in the array. Using boolean or floats as keys might lead to unexpected results. The value corresponding to each key can be of any type, so it is possible to create arrays of arrays, and it is also possible to mix the types for both keys and values (see the next section for some examples). More strictly typed languages require that arrays are defined as lists of the same data type and that the memory must be allocated before the arrays are used.

Objects are usually created as an instance of a class or are generated by the engine, and they will contain methods and/or properties. Properties and methods are accessed with the `->` indirection symbol, for example, `$obj->property` or `$obj->method($a, $b)`.

Resources are a special type that can be created only by the engine (built-in or extension functions). The data structure and memory usage is known only to a few functions used to create, modify, and destroy the resource. It is not possible to convert any other type to a resource type.

Operating in a loosely typed language can make it difficult to know the type of a variable. PHP has a number of functions that can determine the current type of a variable (see Table 10-2).

Table 10-2. *Functions to Check Data Type*

Name	Description
`is_null()`	Returns `true` if the value is null (no type)
`is_string()`	Returns `true` if the value is a string
`is_int()`	Returns `true` if the value is an integer
`is_float()`	Returns `true` if the value is a floating-point value
`is_array()`	Returns `true` if the value is an array
`is_object()`	Returns `true` if the value is an object
`is_a()`	Deprecated; checks if an object is a specified class
`instanceof()`	Checks if an object is an instance of a class

In addition to these functions, two more functions are important when variables are checked. The `isset()` function checks if a variable has been defined, and the `empty()` function checks if the value of a variable is empty. Using one of the `is_*()` functions will give a compiler notice if the variable is undefined. This is not the case for `isset()` and `empty()`. They will return `false` and `true` if the variable is undefined. The next example shows what the `empty()` function will return when passed different values.

The Code

```
<?php
// Example 10-1-1.php
$text = array(
  "0", "1", "\"\"", "\"0\"", "\"1\"",
  "true", "false", "array()", "array(\"1\")"
);
$values = array(0, 1, "", "0", "1", true, false, array(), array("1"));
foreach($values as $i=>$val) {
  echo "empty(" . $text[$i] . ") is " . (empty($val) ? "True" : "False") . "\n";
}
?>
```

How It Works

This example defines two arrays with the same number of elements. The `$text` array prints the values that are checked, and the second array, `$values`, is used in the loop to check the result of a call to the `empty()` function. The output looks like this:

```
empty(0) is True
empty(1) is False
empty("") is True
empty("0") is True
empty("1") is False
empty(true) is False
empty(false) is True
empty(array()) is True
empty(array("1")) is False
```

Note that the values `0`, `""`, `"0"`, and `array()` all are considered empty.

10-2. Assigning and Comparing

Assigning a value to a variable takes place with one of the assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `.=`, `&=`, `|=`, `^=`, `<<=`, or `>>=`. The simple form (`=`) creates a new variable of any type or assigns a new value. The left side is the variable, and the right side is the value or an expression. The remaining assignment types are more complex; they all assume that the variable on the left side is defined before the statement is reached. The result will be the current value of the variable on the left side and the value on the right side after performing the operation identified by the operator. `$a += $b;` is the same as `$a = $a + $b;`.

If the variable is in use when a value is assigned (with simple assignment using the `=` operator), the old value will be discarded before the new variable is created. All the other assignment operators will reuse the existing value to create a new value. If needed, the existing value will be converted to the proper type before the calculation and assignment. For instance, if `$a` is an integer and it is used with the string concatenation operator, then `$a .= "string value";`.

PHP uses a reference-counting system on all variables, so you do not need to free variables when they are no longer used. All allocated memory will be released at the end of the request, but for scripts that use a lot of memory or long-running processes, such as command-line interface (CLI) scripts or PHP-GTK scripts, it might be necessary to free unused variables to allow other variables to use the memory. You can free any variable from memory by assigning it to `NULL (`$a = NULL;`)` or by using the `unset()` function.

■ **Note** If more than one variable name references the same variable, all of them must be unset before the memory is released. Creating multiple references to the same data in memory is discussed in this recipe.

You can add values to arrays in two ways. If the left side is a variable, the right side can be an array definition like this: `$a = array(9, 7, "orange", "apple");`. This will create an array with four elements, and the index or key values will be assigned automatically in numeric order starting with 0. New values can be added, or existing values can be replaced with an expression where the left side points to one of the values in the array. So, setting `$a[2] = "pear";` will replace the third element, `orange`, with `pear` because the key value of 2 was in use already. A new element will be added to the array if the key does not exist already. Setting `$a[5] = "orange";` will add `orange` with the key 5, and the array will now have five elements. Note that this will not have an element with the key 4. If you try to access or use `$a[4]`, you will get an undefined variable notice. You can use a special notation to let PHP assign the key values automatically. You do this by simply omitting the key in the assignment, such as `$a[] = "apricot"`. This will create the key 6 and assign it the value `apricot`. This notation will always use numeric indexes, and the next value will be one higher than the highest numeric index value in the array.

You can also assign the key values to force a specific relation between keys and values, as shown the following example, where both keys and values are mixed between numeric and string values.

The Code

```
<?php
// Example 10-2-1.php
$a = array(
  0=>1,
  1=>2,
  2=>"orange",
  3=>"apple",
  "id"=>7,
  "name"=>"John Smith"
);
print_r($a);
?>
```

How It Works

In this example you create an array with six values where the keys are assigned with the `=>` operator. The first four values are assigned numeric keys, and the last two are assigned string keys. The output from this code looks like this:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => orange
    [3] => apple
    [id] => 7
    [name] => John Smith
)
```

You can get rid of a single value in an array with the `unset()` function. This will remove the value from the array but not rearrange any of the key values. The code `unset($a[3]);` will remove `apple` from the array in the previous example. PHP implements many functions that manipulate arrays. One of these requires special attention. It is the `list()` function, or language construct. Like `array()`, it is not really a function but a way to tell the engine how to handle special data. It is used on the left side of the assignment operator, when the right side is an array or an expression that results in an array, and it can assign values to multiple variables at the same time.

■ **Note** `list()` works only on numerical arrays and assumes numerical indexes start at 0.

The next example shows how to use the `list()` function.

The Code

```
<?php
// Example 10-2-2.php
$net_address = array("192.168.1.101", "255.255.255.0", "192.168.1.1");
list($ip_addr, $net_mask, $gateway) = $net_address;
echo "ip addr  = $ip_addr\n";
echo "net mask = $net_mask\n";
echo "gateway  = $gateway\n";
?>
```

How It Works

First, you define an array with three elements. This could be the return value from a function call. Second, these values are extracted from the array and stored in individual variables with a call to the `list()` function. Finally, the three new variables are printed to form this output:

```
ip addr  = 192.168.1.101
net mask = 255.255.255.0
gateway  = 192.168.1.1
```

When a variable is assigned a value, it will actually get a copy of that value. Using the special & operator makes it possible to create a new variable that references the same value in memory as another variable. This is best demonstrated with a small example, where two values are defined. In the first part of the code, `$b` is assigned a copy of `$a`, and in the second part, `$b` is assigned a reference to `$a`.

The Code

```
<?php
// Example 10-2-3.php
$a = 5;
$b = $a;
$a = 7;
echo "\$a = $a and \$b = $b\n";

$a = 5;
$b = &$a;
$a = 7;
echo "\$a = $a and \$b = $b\n";
?>
```

How It Works

In the first part, `$a` and `$b` will have independent values, so changing one variable will not affect the other. In the second part, the two variables share the same memory, so changing one variable will affect the value of the other.

```
$a = 7 and $b = 5
$a = 7 and $b = 7
```

When two or more variables share the same memory, it is possible to use the `unset()` function on one of the variables without affecting the other variables. The `unset()` function will simply remove the reference and not the value.

PHP has two kinds of comparison operators. The loose comparison operators will compare values even if the two values are of different data types. The strict comparison operators will compare both the values and the data types. So, if two variables are of different types, they will always be different when compared to the strict operators, even if the values are identical otherwise. Tables 10-3 and 10-4 explain the comparison operators.

Table 10-3. *Loose Comparison Operators*

Example	Name	Description
`$a == $b`	Equal to	`True` if `$a` is equal to `$b`
`$a != $b`	Not equal to	`True` if `$a` is not equal to `$b`
`$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`

Table 10-4. *Strict Comparison Operators*

Example	Name	Description
`$a === $b`	Equal to	True if $a is equal to $b and they are of the same type
`$a !== $b`	Not equal to	True if $a is not equal to $b or they are not of the same type

When the loose operators are used and the data types are different, PHP will convert one of the variables to the same type as the other before making the comparison.

To show how these different operators work, the next example creates a script that loops through an array of different data types and compares all the values to each other.

The Code

```
<?php
// Example 10-2-4.php
$Values = array(
  NULL,
  True,
  False,
  1,
  0,
  1.0,
  0.0,
  "1",
  "0",
  array(1),
  (object)array(1)
);

function dump_value($var) {
  switch (gettype($var)) {
    case 'NULL':
      return "NULL";
      break;
    case 'boolean':
      return $var ? "True" : "False";
      break;
    default :
    case 'integer':
      return $var;
      break;
    case 'double':
      return sprintf("%0.1f", $var);
      break;
    case 'string':
      return "'$var'";
      break;
```

```
    case 'object':
    case 'array':
      return gettype($var);
      break;
  }
}

function CreateTable($Values, $type = "==") {
  echo "<table border=1>";
  echo "<tr><td>$type</td>";
    foreach ($Values as $x_val) {
      echo "<td bgcolor=lightgrey>" . dump_value($x_val) . "</td>";
    }
  echo "</tr>";
  foreach ($Values as $y_val) {
    echo "<tr><td bgcolor=lightgrey>" . dump_value($y_val) . "</td>";
    foreach ($Values as $x_val) {
      if ($type == "==") {
        $result = dump_value($y_val == $x_val);
      }
      else {
        $result = dump_value($y_val === $x_val);
      }
      echo "<td>$result</td>";
    }
    echo "</tr>";
  }
  echo "</table>";
}

echo "<html><body>";
CreateTable($Values, "==");
CreateTable($Values, "===");
echo "</body></html>";
?>
```

How It Works

The script defines the array with values of different types, a function to format the output, and a function to create a Hypertext Markup Language (HTML) table with the result. The formatting function `dump_value()` is needed to print readable values for booleans and floats. The `CreateTable()` function is called once for each comparison type. The output from this script, viewed in a browser, looks like Figure 10-1 and Figure 10-2.

==	NULL	True	False	1	0	1.0	0.0	'1'	'0'	array	object
NULL	True	False	True	False	True	False	True	False	False	False	False
True	False	True	False	True	False	True	False	True	False	True	True
False	True	False	True	False	True	False	True	False	True	False	False
1	False	True	False	True	False	True	False	True	False	False	False
0	True	False	True	False	True	False	True	False	True	False	False
1.0	False	True	False	True	False	True	False	True	False	False	False
0.0	True	False	True	False	True	False	True	False	True	False	False
'1'	False	True	False	True	False	True	False	True	False	False	False
'0'	False	False	True	False	True	False	True	False	True	False	False
array	False	True	False	False	False	False	False	False	False	True	False
object	False	True	False	False	False	False	False	False	False	False	True

Figure 10-1. *Comparing variables of different types with loose operators*

===	NULL	True	False	1	0	1.0	0.0	'1'	'0'	array	object
NULL	True	False	False	False	False	False	False	False	False	False	False
True	False	True	False	False	False	False	False	False	False	False	False
False	False	False	True	False	False	False	False	False	False	False	False
1	False	False	False	True	False	False	False	False	False	False	False
0	False	False	False	False	True	False	False	False	False	False	False
1.0	False	False	False	False	False	True	False	False	False	False	False
0.0	False	False	False	False	False	False	True	False	False	False	False
'1'	False	False	False	False	False	False	False	True	False	False	False
'0'	False	False	False	False	False	False	False	False	True	False	False
array	False	False	False	False	False	False	False	False	False	True	False
object	False	False	False	False	False	False	False	False	False	False	True

Figure 10-2. *Comparing variables of different types with strict operators*

10-3. Typecasting

Typecasting is a method used to force the conversion of a variable from one type to another. During typecasting, the value is preserved and converted if possible, or the result is assigned a default value with the specified type. Converting a string with `abc` to an integer will give the value 0. The next example shows how a string with a numeric value can be typecast to an integer and how an array, which has at least one element, is typecast to an integer that will result in a value of 1.

The Code

```
<?php
// Example 10-3-1.php
$a = "10";
$b = (int)$a;
echo 'gettype($a) = ' . gettype($a) . "\n";
echo 'gettype($b) = ' . gettype($b) . ", \$b = $b\n";
$a = array(5,4,5);
$b = (int)$a;
echo 'gettype($a) = ' . gettype($a) . "\n";
echo 'gettype($b) = ' . gettype($b) . ", \$b = $b\n";
?>
```

How It Works

You define `$a` as a string and then `$b` as the integer value of `$a`. Then you use the `gettype()` function to get a string representation of the variable type. The output from this script looks like this:

```
gettype($a) = string
gettype($b) = integer, $b = 10
gettype($a) = array
gettype($b) = integer, $b = 1
```

■ **Note** Converting from arrays and objects to integers is undefined by the engine, but it currently works as if the variable was converted to a boolean and then to an integer. You should not rely on this, and you should avoid typecasting arrays and objects to any other types.

When arrays are used with an `if` clause, they are implicitly converted to booleans. This is useful when checking if an array has any elements. If `$a` is an array, then the code `if ($a) echo "$a has elements";` will print a statement only if `$a` is a nonempty array.

Jon Stephen's Chapter 4 discussed numeric values and showed how an integer value could change its type to floating point if the result of a calculation was outside the boundaries of an integer.

In this chapter you have seen how you can convert string values with numeric content into integers. You can apply the same conversion to floating-point values but not to boolean values. For example, `(bool)"true";` and `(bool)"false";` will both return a `true` value. An empty string will convert to `false`, and any nonempty string will convert to `true` when typecast to a boolean.

It is also possible to convert variables from arrays to objects and back again. You can do this to change how elements/properties are accessed, as shown in the following example.

The Code

```
<?php
// Example 10-3-2.php
$a = array(
  "Name" => "John Smith",
  "Address" => "22 Main Street",
  "City" => "Irvine",
  "State" => "CA",
  "Zip" => "92618"
);
echo "Name = " . $a["Name"] . "\n";

$o = (object)$a;
echo "Address = $o->Address\n";?>
```

How It Works

First, you define an array with five elements. Each element is defined as a key and a value, and all the keys are string values. Second, you use traditional array accessors to print the `Name` value from the array. Finally, a new variable is created by typecasting the array to an object. When elements/properties are accessed on an object, you use the `->` symbol between the object name and the property.

```
Name = John Smith
Address = 22 Main Street
```

Converting an object to an array will convert properties to elements of the resulting array only (see recipe 10-5 for a discussion of the `public`, `private`, and `protected` properties).

The Code

```
<?php
// Example 10-3-3.php
class myclass {
  public $name;
  public $address;
  private $age;
  function SetAge($age) {
    $this->age = $age;
  }
}

$obj = new myclass;
$obj->name = "John Smith";
$obj->address = "22 Main Street";
$obj->SetAge(47);

$arr = (array)$obj;
print_r($arr);
?>
```

How It Works

The class `myclass()` has a couple of `public` properties, a `private` property, and a method used to set the `private` property. When an object is created as an instance of `myclass`, you can use `->` to assign values to the `public` properties and use the `SetAge()` method to assign a value to the `private` property. The object is then converted to an array and dumped with the `print_r()` function.

```
Array
(
    [name] => John Smith
     [address] => 22 Main Street
    [ myclass age] => 47
)
```

Formatting output requires different types to be converted into strings before they are sent to the client. You can do this by concatenating different values using the `.` operator. The engine will automatically convert nonstring values to strings, if possible. Integer and floating-point values are converted into a decimal representation, and booleans are converted into an empty value or 1.

■ **Note** If an expression is concatenated with other values or strings, you must enclose the expression in `( )`. For instance, `$a = "test " . 5 + 7;` is not the same as `$a = "test " . (5 + 7);`. The first will calculate to the value 7, as the concatenation will take place before the addition, so the string `"test 5"` is created and added to the value 7. The second expression will calculate to `"test 12"`.

Arrays, objects, and resources contain values too complex to be converted to strings in a unified and automated way, so these are converted into strings showing the data type.

It is also possible to embed variables directly into strings, when the string is created with double quotes. A string with single quotes will not expand the value of any variable included in the string. The next example shows how embedded variables are handled when the string is created with single or double quotes.

The Code

```
<?php
// Example 10-3-4.php
$a = 10;
$b = 15.7;
echo "The value of \$a is $a and the value of \$b is $b\n";
echo 'The value of \$a is $a and the value of \$b is $b\n';
?>
```

How It Works

This example will output two lines, where the first line will expand the values of $a and $b and where the variable names are printed in the second line. The \ escapes the $ signs to prevent the engine from converting the first $a into the value, and it just prints the variable name. Note how the string with single quotes prints all the escape characters.

```
The value of $a is 10 and the value of $b is 15.7
The value of \$a is $a and the value of \$b is $b\n
```

The same example with the concatenation operator looks like the following.

The Code

```
<?php
// Example 10-3-5.php
$a = 10;
$b = 15.7;
echo "The value of \$a is " . $a . " and the value of \$b is " . $b . "\n";
echo 'The value of $a is ' . $a . ' and the value of $b is ' . $b . "\n";
?>
```

How It Works

Note how the last line combines strings created with single and double quotes. This allows you to use $a without escaping the $ sign and the new line at the end of the line.

Embedding numbers and strings into other strings is simple, but what if the value is stored in an array or object? It is still possible to embed these more complex types in strings, but you need to follow a few rules:

- You can use only one dimension.
- You should not include key values in quotes, even if strings are used as keys.
- You can embed more complex values with the syntax ${}.

The next example shows how arrays embedded in strings will be converted.

The Code

```
<?php
// Example 10-3-6.php
$arr = array(
  1 => "abc",
  "abc" => 123.5,
  array(1,2,3)
);
$key = "abc";
```

```
echo "First value = $arr[1]\n";
echo "Second value = $arr[abc]\n";
echo "Third value = $arr[2]\n";
echo "Third value = $arr[2][2]\n";

echo "Second value = ${arr['abc']}\n";
echo "Second value = ${arr["abc"]}\n";
echo "Second value = ${arr[$key]}\n";
?>
```

How It Works

After defining an array with three elements and a string value with the index of one of the elements, you use the different embedding methods to see how the values are resolved. The three first lines in the output, shown next, shows how the simple embedding works. The first two of these actually print the value of the element, but the third line prints `Array`. The same goes for the fourth line where you tried to print a single value from a two-dimensional array. The last three lines used the `${}` syntax that allows embedding of more complex types, but this is limited to one-dimensional arrays. Use string concatenation if you want to combine values from multidimensional arrays in a string.

```
First value = abc
Second value = 123.5
Third value = Array
Third value = Array[2]
Second value = 123.5
Second value = 123.5
Second value = 123.5
```

The following example is the same but with objects.

The Code

```
<?php
// Example 10-3-7.php
$arr = array(
  "abc" => "abc",
  "def" => 123.5,
  "ghi" => array(1,2,3)
);
$key = "abc";
$obj = (object) $arr;

echo "First value = $obj->abc\n";
echo "Second value = $obj->def\n";
echo "Third value = $obj->ghi\n";
?>
```

```
First value = abc
Second value = 123.5
Third value = Array
```

■ **Note** It is important that the index values of the array are strings. Values that use an integer as an index cannot be converted to a valid property name. Variable and property names must start with a letter or an underscore.

10-4. Using Constants

You can use variables to define values that have one value for the duration of the script. The nature of a variable allows the content to be changed, and this might lead to unexpected behavior of the program. This is where constants become handy. Constants are identifiers for simple values. The value can be defined once, while the script is running, and never changed. The function `define()` assigns a simple constant value (`bool`, `int`, `float`, or `string`) to a constant name. By default the constant names are case-sensitive like variables, but a third optional argument to the `define()` function makes it possible to create case-insensitive constant names. Constant names are often defined as uppercase only to make it easier to identify them in the code. The `define()` function will return `true` if the constant could be defined or `false` if it was defined already.

Unlike variables that start with a `$` sign, constants are defined by name; this makes it impossible for the engine to identify constants with the same name as language constructs or functions. If a constant is defined with a name that is reserved for language constructs or function names, it can be retrieved only with the `constant()` function. This function takes a string as the argument and returns the value of the constant. The `constant()` function is also helpful when different constants are retrieved by storing the constant name in a variable or returning it from a function.

The Code

```
<?php
// Example 10-4-1.php
define('ALIGN_LEFT', 'left');
define('ALIGN_RIGHT', 'right');
define('ALIGN_CENTER', 'center');

$const = 'ALIGN_CENTER';
echo constant($const);
?>
```

How It Works

This example defines three constants and assigns the name of one of the constants to a string that is used as the parameter to the `constant()` function. The result is the value of the constant.

```
center
```

You can use the function `defined()` to check if a constant is defined, before trying to define it again or before using it to avoid undefined constants (which will generate a warning).

Using constants makes it easy to change the values used to control program flow without having to break code. If you use hard-coded values and want to change one or more values, you must make sure all the places you compare to each value are updated to match the new values. If, on the other hand, you use constants, then you can get by with changing the value in the constant definition, and all the places you use that constant will automatically have the new value.

Consider an example where you have three values controlling the program flow and you want to change the values for some reason. Your code could look like the following example.

The Code

```
<?php
// Example 10-4-2.php
switch($justify) {
  case 1 :  // left
    break;
  case 2 :  // center
    break;
  case 3 :  // right
    break;
}
?>
```

How It Works

Each constant is used only once in the example, but you could have several functions that use a justification value to print the content in different ways, and using numbers is less readable than the constant names.

The Code

```
<?php
// Example 10-4-3.php
define('ALIGN_LEFT', 1);
define('ALIGN_CENTER', 2);
define('ALIGN_RIGHT', 3);

switch($value) {
  case ALIGN_LEFT :
    break;
  case ALIGN_CENTER :
    break;
  case ALIGN_RIGHT :
    break;
}
?>
```

How It Works

So, to change the values of these constants, you need to change only the definitions, and you get the benefit of writing more readable code without having to add a lot of comments.

PHP has a large number of predefined constants (`M_PI`, `M_E`, and so on, from the math functions), and many extensions define and use constants (`MYSQL_NUM`, `MYSQL_ASSOC`, and `MYSQL_BOTH`, to mention a few) that allow you to write more readable code.

It is not possible to define a constant as an array or object, but as discussed in recipe 10-4, you can convert these data types into strings with the `serialize()` function. You can use the result of this function, or any other function that returns a simple value, to define constant values. These constants can then be accessed globally (as discussed in recipe 10-5). The only downside is the need to unserialize the value before it can be used. The next example shows how to use this technique to store an array in a constant and use that from within a function. This makes it possible to access a global constant in the form of an array, without having to use `global $arr;` or `$GLOBALS['arr'];`.

The Code

```
<?php
// Example 10-4-4.php
$arr = array("apple", "orange", "pear");
define('MYARRAY', serialize($arr));

function MyTest() {
  print_r(unserialize(MYARRAY));
}

MyTest();
?>
```

How It Works

The variable `$arr` is assigned an array with three values, serialized (converted to string form), and stored in a constant called `MYARRAY`. The constant is then used inside the function `MyTest()`, where it is converted back to an array and the content is printed. The output looks like this:

```
Array
(
    [0] => apple
    [1] => orange
    [2] => pear
)
```

10-5. Defining Variable Scope

Variables are visible and usable in the scope where they are defined, so if a variable is defined in the global scope, it is visible there and not in any functions or class methods. If the variable `$a` is defined globally, another variable with the same name might be defined in a function. The two variables are not the same even though they share the same name.

The Code

```
<?php
// Example 10-5-1.php
$a = 7;
function test() {
  $a = 20;
}
test();
echo "\$a = $a\n";
?>
```

How It Works

The variable `$a` is defined in the global scope and assigned the value 7. Inside the function `test()` you define another variable with the same name but the value 20. When the code is executed, you call the function `test` and then print the value of `$a`. The two versions of `$a` do not share the same memory, so the output will be the original value of `$a` from the global scope.

```
$a = 7
```

You have two ways to access global variables from within a function or method of a class. You can use the `global` keyword to associate a variable inside a function with a global variable. The variable does not need to be defined globally before the association is made, so if the line `$a = 7;` in the following example is omitted, the result will still be 20.

The Code

```
<?php
// Example 10-5-2.php
$a = 7;
function test() {
  global $a;
  $a = 20;
}
test();
echo "\$a = $a\n";
?>
```

How It Works

The only change from the previous example is the line `global a$;` inside the function. This line makes the two variables reference the same memory, so when you change the value inside the function, you also change the value of the variable in the global scope.

```
$a = 20
```

The other way of accessing global variables is by using the true global or superglobal variable called `$GLOBALS`. This is an associative array that is available in any scope, and it has references to all variables defined in the global scope.

The Code

```
<?php
// Example 10-5-3.php
$a = 7;
function test() {
  $GLOBALS['a'] = 20;
}
test();
echo "\$a = $a\n";
?>
```

How It Works

By using the superglobal `$GLOBAL`, it is possible to access or change any variable from the global space, without defining it as global as you did in the previous example.

```
$a = 20
```

As in the previous example, it is possible to define variables in the global scope from within a function or class method. Using `$GLOBALS['newvar'] = 'test';` will create a variable called `$newvar` in the global scope and assign it the string value `'test'`.

You can use a few other PHP variables like this. These are in general called *superglobals*, and they do not belong to any special scope (see Table 10-5).

Table 10-5. *PHP Superglobals*

Name	Description
$GLOBALS	An associated array with references to every variable defined in the global scope
$_SERVER	Variables set by the server
$_ENV	Environment variables
$_GET	Variables provided to the script via the Uniform Resource Locator (URL)
$_POST	Variables provided to the script via HTTP POST
$_COOKIE	Variables provided to the script via HTTP cookies
$_FILE	Variables uploaded via HTTP POST file uploads
$_REQUEST	A combination of variables provided by GET, POST, and COOKIE methods
$_SESSION	Variables currently registered in the session

Constants are another form of true global data. If a script has a need for defining values that should be accessed from any scope, constants might be a good way of defining these. This, of course, requires that the values should remain constant for the duration of the script. You can define constants in the global scope or in a function, but they will always belong to the global scope, as shown in the next example.

The Code

```
<?php
// Example 10-5-4.php
define('CONST1', 1);

function MyTest() {
  define('CONST2', 2);
}

MyTest();
echo "CONST1 = " . CONST1 . " and CONST2 = " . CONST2 . "\n";
?>
```

How It Works

In this example, you define a constant from the global scope and one from inside a function. As the output shows, both constants are available in the global scope.

```
CONST1 = 1 and CONST2 = 2
```

Working with classes and objects introduces another form of variable called a *property*, or a *member*. This is basically a normal PHP variable, but access to it can be restricted with one of the keywords `public`, `private`, `protected`, or `static`. You can use the same keywords when declaring functions or methods. Older versions of PHP (before version 5.*x*) used `var` to declare members, and they were all considered to be public. When updating scripts from PHP 4 to PHP 5, you should convert all `var` declarations to one of the new modifiers. Table 10-6 lists the class member and method definitions.

Table 10-6. *Class Member and Method Definitions*

Name	Description
`Const`	Defines a constant member.
`Public`	Accessible from any object of the class.
`Protected`	Accessible from the class where it is defined and from inherited classes.
`Private`	Accessible from the class where it is defined.
`Static`	Modifier. When used alone, public is assumed.

The Code

```
<?php
// Example 10-5-5.php
class myclass {
  public $a;

  function set_value($val) {
    $this->a = $val;
  }
}

$obj = new myclass;
$obj->set_value(123);
echo "Member a = $obj->a\n";
$obj->a = 7;
echo "Member a = $obj->a\n";
?>
```

How It Works

This example declares a class called `myclass()`. It has the public member `$a` and a method called `set_value()`. An object is defined as an instance of `myclass()`, and then you use the `set_value()` method to assign a value to the member. This value is later changed by accessing the member directly.

```
Member a = 123
Member a = 7
```

Changing the member `$a` to `protected` or `private` will give the following result.

The Code

```
<?php
// Example 10-5-6.php
class myclass {
  private $a;

  function set_value($val) {
    $this->a = $val;
  }
}

$obj = new myclass;
$obj->set_value(123);
echo "Member a = $obj->a\n";
$obj->a = 7;
echo "Member a = $obj->a\n";
?>
```

How It Works

This small change will cause the script to fail.

```
Fatal error: Cannot access private property myclass::$a
in /Samples/11-5-5.php on line 12
```

This feature is useful when you develop classes that are used by other developers. It will protect the class from being misused by accessing the members directly for both reading and writing. The class should expose functions to set and get values that are supposed to be available (the class API) to other developers. So, you should modify this class as shown in the following example.

The Code

```
<?php
// Example 10-5-7.php
class myclass {
  private $a;

  function set_value($val) {
    $this->a = $val;
  }

  function get_value() {
    return $this->a;
  }
}
```

```
$obj = new myclass;
$obj->set_value(123);
echo "Member a = " . $obj->get_value() . "\n";
?>
```

How It Works

You can access the member `$a` only through one of the methods.

```
Member a = 123
```

This will allow read and write access to the member but will not allow direct access to modify the member without calling a method. The method should check the value and return a value indicating if the property could be set. If a member is `private`, it can be accessed only by members of the class where it is created; if a member is `protected`, it can be modified only by the class or any inherited classes.

You can use the static modifier to change a member or method so it is accessible without instantiating the class. A static member will be defined only once regardless of the number of instantiated objects of the class.

The Code

```
<?php
// Example 10-5-8.php
class myclass {
  const MYCONST = 123;
  static $value = 567;
}

echo 'myclass::MYCONST = ' . myclass::MYCONST . "\n";
echo 'myclass::$value = ' . myclass::$value . "\n";
?>
```

How It Works

In this example, a simple class defines two members. One is defined as a `const`, and the other is defined as a `static`. Both members can be accessed with the name of the class and two colons and the name of the member. As for normal PHP constants, the `const` members of a class are read-only.

```
myclass::MYCONST = 123
myclass::$value = 567
```

Note how the constant definition automatically is considered a static member of the class (only one copy will be stored in memory for all instances of the class) and how the `static` modifier is used without a `public`, `private`, or `protected` keyword. This makes the variable public. If the variable was defined as `private static`, it would not be possible to access it directly, as shown in the next example.

The Code

```
<?php
// Example 10-5-9.php
class myclass {
  const MYCONST = 123;
  private static $value = 567;
}

echo 'myclass::MYCONST = ' . myclass::MYCONST . "\n";
echo 'myclass::$value = ' . myclass::$value . "\n";
?>
```

How It Works

The first part of the code works as in the previous example, but when you try to access the private member, the script will stop with a fatal error.

```
myclass::MYCONST = 123
Fatal error: Cannot access private property myclass::$value
in /Samples/10-5-9.php on line 9
```

10-6. Parsing Values to Functions

The function name and the number of parameters it takes define a function. Each parameter can be defined as *pass by value* or *pass by reference* or can be assigned a default value. Using default values makes it possible to call the function with fewer arguments, and parameters with default values should always be placed at the end of the parameter list.

When a variable is passed by value, it means that the function will operate on a copy of the variable. The function can change the content and type of the variable without affecting the code that called the function (that is, that passed the argument). If a variable is passed by reference, it means that the variable will share the same memory, and any changes to the content or type will affect the code that called the function. The next example shows two functions that both take one variable as a parameter.

The Code

```
<?php
// Example 10-6-1.php
function by_value($a) {
  $a *= 2;
}
function by_reference(&$a) {
  $a *= 2;
}
$b = 5;
by_value($b);
```

```
echo "\$b is now $b\n";
by_reference($b);
echo "\$b is now $b\n";
by_value(&$b);
echo "\$b is now $b\n";
?>
```

How It Works

The two functions are almost identical. They both take the value passed as the argument and multiply by 2. The difference is how the variable is passed. In the first function, the variable is passed by value, so `$a` is considered a copy of the variable. The second function forces the variable to be passed by reference. This makes the two variables share the same memory; therefore, when the variable is changed inside the function, it affects the variable that was passed. You can force parsing by reference at call time. You do this by adding the `&` sign in front of the variable name. The output from this example looks like this:

```
$b is now 5
$b is now 10
$b is now 20
```

Passing values by reference is a useful way to have a function return more than one value. A function that performs a database query to get a result set could also return information about the columns selected, and the actual return value could be used to indicate success or failure. To illustrate this, create an example with two functions. `GetData()` simulates a database query, and `ListData()` creates an HTML table with the rows returned from `GetData()`. You can also extend this example to include another function to present data, when only a single row is returned from the `GetData()` function.

The Code

```
<?php
// Example 10-6-2.php
define('COLUMN_NAME', 0);
define('COLUMN_TYPE', 1);

define('COLUMN_STRING', 1);
define('COLUMN_INTEGER', 2);

function GetData(&$data, &$meta) {
  $meta = array(
    array(
      COLUMN_NAME => "First Name",
      COLUMN_TYPE => COLUMN_STRING
    ),
```

```
    array(
      COLUMN_NAME => "Last Name",
      COLUMN_TYPE => COLUMN_STRING
    ),
    array(
      COLUMN_NAME => "Age",
      COLUMN_TYPE => COLUMN_INTEGER
    )
  );
  $data = array(
    array("John", "Smith", 55),
    array("Mike", "Johnson", 33),
    array("Susan", "Donovan", 29),
    array("King", "Tut", 3346)
  );
  return sizeof($data);
}

function ListData($data, $meta) {
  echo "<table border=1>";
  foreach($data as $row) {
    echo "<tr>";
    foreach($row as $col=>$cell) {
      switch ($meta[$col][COLUMN_TYPE]) {
        case COLUMN_STRING :
          echo "<td align=left>$cell</td>";
        break;
      case COLUMN_INTEGER :
          echo "<td align=right>" . number_format($cell) . "</td>";
        break;
      }
    }
    echo "</tr>";
  }
  echo "</table>";
}

$d = array();
$m = array();
if (GetData($d, $m)) {
  ListData($d, $m);
}
?>
```

How It Works

You define the two variables `$d` and `$m`. Both are assigned the value of empty arrays. The call to `GetData()` defines the content of the two variables passed by reference. The two variables are then passed to the `ListData()` function, which generates an HTML table showing the values, as shown in Figure 10-3.

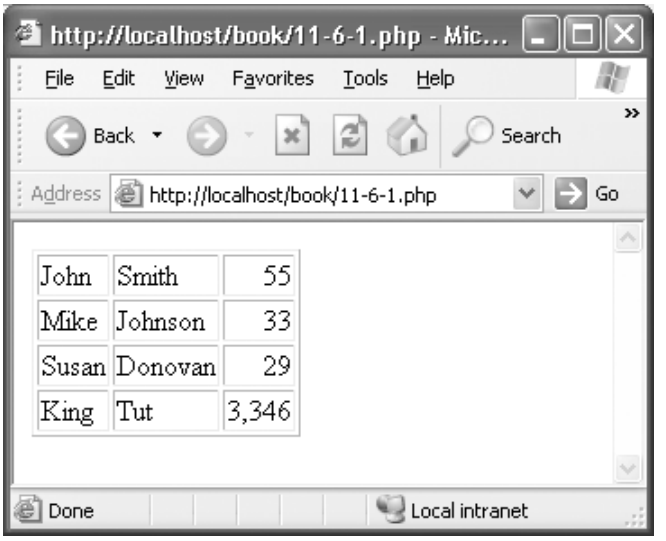

Figure 10-3. *Getting and listing data*

A special case of pass by value is used for arrays. Arrays can be very large, and in order to improve speed these values are always passed by reference. If the definition is called for pass by value, the array will be copied when the function first attempts to modify the content or data type. This is called *copy on write*, so if an array is passed by value and the function never changes the content of the array, you have no need to perform the copy.

You can define whether a variable is passed by value or reference either in the function definition or when the function is called, as shown in the following example.

The Code

```
<?php
// Example 10-6-3.php
function f1($a) {
  $a += 4;
}
function f2(&$a) {
  $a += 10;
}
$b = 5;
f1(&$b);
```

```
f2($b);
echo "\$b = $b\n";
?>
```

How It Works

This example defines two functions. The first function, `f1()`, takes an argument passed by value, and the second function, `f2()`, takes one argument passed by reference. When the first function is called, the value that is passed is a reference to `$b`, forcing the function to operate on the same value in memory. When the second function is called, the value passed is the actual value, but the function automatically converts that to a reference to the value.

```
$b = 19
```

10-7. Using Dynamic Variable and Function Names

You can use variable variables or variable function names to reduce the number of `if`, `else`, or `switch` statements and make the code more readable. It is all about being able to calculate the name of the variable to store (or get data from) or the name of the function to execute.

Calculating the index or key value for an array is useful if the data is stored in an array. You can specify the key value with a hard-coded value or with a value stored in a variable or returned from a function.

The Code

```
<?php
// Example 10-7-1.php
$fruits = array(
  'apple', 'orange', 'pear', 'apricot',
  'apple', 'apricot', 'orange', 'orange'
);
$fruit_count = array();
foreach ($fruits as $i=>$fruit) {
  if (isset($fruit_count[$fruit])) {
    $fruit_count[$fruit]++;
  }
  else {
    $fruit_count[$fruit] = 1;
  }
}
asort($fruit_count);
foreach ($fruit_count as $fruit=>$count) {
  echo "$fruit = $count\n";
}
?>
```

How It Works

The script produces this output:

```
pear = 1
apple = 2
apricot = 2
orange = 3
```

This example loops through an indexed array of fruit names and creates a new array with the count of each fruit name. The `$fruit_count` array is filled with key and value pairs as you loop through the `$fruits` array. For each fruit you test to see if it is a new name or if it already exists in the array. This code can be written a little more compactly by using the `@` modifier. This will suppress any warnings from using the increment operator (`++`) on an undefined variable. If a variable is undefined when the increment operator is used, a new variable will be declared with a 0 value, and a warning will be issued. This warning can be suppressed by adding `@` in front of the statement. You can also use the `@` modifier to suppress warnings from function calls, but this will not suppress errors.

```
<?php
// Example 10-7-1a.php
$fruits = array(
  'apple', 'orange', 'pear', 'apricot',
  'apple', 'apricot', 'orange', 'orange'
);
$fruit_count = array();
foreach ($fruits as $i=>$fruit) {
  @$fruit_count[$fruit]++;
}
asort($fruit_count);
foreach ($fruit_count as $fruit=>$count) {
  echo "$fruit = $count\n";
}
?>
```

■ **Caution** Using the `@` modifier in front of variables or functions could hide warnings that may indicate a programming error. You should use it with caution.

It is also possible to calculate the variable name for simpler variables and use that name to access the value of that variable. You do this with the double `$` sign. Adding another `$` sign in front of a variable will take the value of that variable and access the value of another variable with that name. If `$a = 'test';`, then `$$a` will access a variable called `$test`. The following example shows how a series of variables is accessed to print a string composed from the values of these variables.

The Code

```
<?php
// Example 10-7-2.php
$a0 = 'This';
$a1 = 'is';
$a2 = 'a';
$a3 = 'test';

for ($i = 0; $i < 4; $i++) {
  $var = "a$i";
  echo “${$var} ";
}
?>
```

How It Works

The script defines four variables that all start with `$a` and end with an integer. You then create a loop from 0 to 3, and for each execution of the loop you output the value of the variable with the name calculated from the contents of the string `$var`.

```
This is a test
```

The calculation of each variable is simple, but you could easily extend the same method to include more advanced calculations or database lookups.

■ **Note** When a variable is embedded in a string, it is necessary to use a different notation to avoid errors. `$$var` becomes `${$var}` when it is embedded in a string.

You can also use constants in the calculation of variable variables. You need to use a different modifier, because a $ sign in front of a constant would look like a variable. By putting {} around the constant name and then applying the $ sign in front of it, you will create a reference to a variable with the name of the constant's value.

The Code

```
<?php
// Example 10-7-3.php
define('CONST_A', 'test');

${CONST_A} = 27;
echo "\$test = $test\n";
?>
```

How It Works

First you define a variable with the value test, and then you use that constant to calculate a new variable name and assign that variable the value of 27. To avoid creating a new variable called `$CONST_A`, you use the extended notation `${CONST_A}` to tell the engine to use the constant.

```
$test = 27
```

It is also possible to use the value of a variable to point to a function name and thereby change the program flow without needing flow control. This might not always make the code readable, but it makes it possible to create code where the flow control can be moved to a database in the form of parameters.

The Code

```
<?php
// Example 10-7-4.php
function ShowSimple($val) {
  echo "$val\n";
}
function ShowComplex($val) {
  echo "The value is " . number_format($val) . "\n";
}

$v = 1234567;

$a = "ShowSimple";
$b = "ShowComplex";

$a($v);
$b($v);
?>
```

How It Works

You define two functions and assign the names of each function to a variable. When the new variables are written as `$a($v);`, the system will convert `$a` to a function name and call that function.

```
1234567
The value is 1,234,567
```

You can use this method to return the function name from a function call or calculation. It might make the code less readable, and you can obtain the same effect by adding an extra parameter to one function so it will be able to handle both simple and complex printing.

10-8. Encapsulating Complex Data Types

You can format numbers and strings and use them as output or store them in files or databases without modifications. The more complex data types—arrays and objects—can also be stored, but that generally requires some advanced formatting or multiple records in the database (one for each element in the array). This was demonstrated in recipe 10-6, where one function generated multiple arrays and another function presented the generated data in an HTML table structure.

However, using user-defined functions to convert arrays and objects into data that can be stored in a database or file is not the fastest or simplest solution. This is where the built-in functions `serialize()` and `unsearialize()` become handy. These functions can convert an array or an object into a string representation that can be stored in a single column in a database (or a file) and later retrieved and converted to the original data type.

The `serialize()` function takes a PHP variable and converts it into a string representation, and the `unserialize()` function takes a string (most often created with `serialize()`) and converts it to its original type.

■ **Note** Variables of the resource type cannot be serialized. They contain data created and maintained by the engine. Any other type can be serialized.

The Code

```
<?php
// Example 10-8-1.php
$fruits = array(
  'apple', 'orange', 'pear', 'apricot',
  'apple', 'apricot', 'orange', 'orange'
);

$str = serialize($fruits);
echo "$str\n";

$new_fruits = unserialize($str);
$new_fruits[] = 'apple';
print_r($new_fruits);
?>
```

How It Works

This example uses the `serialize()` function to convert the contents of an array to a string. The string is printed and then converted to a new array, where you add a new element.

```
a:8:{i:0;s:5:"apple";i:1;s:6:"orange";i:2;s:4:"pear";i:3;s:7:"apricot";i:4;
s:5:"apple";i:5;s:7:"apricot";i:6;s:6:"orange";i:7;s:6:"orange";}
Array
(
    [0] => apple
    [1] => orange
    [2] => pear
    [3] => apricot
    [4] => apple
    [5] => apricot
    [6] => orange
    [7] => orange
    [8] => apple
)
```

Database-driven websites are often designed in a way so the content stored in a database can be retrieved and presented without much formatting. When a web page is created from numeric data and processing, it can be useful to cache the results for easy access and presentation for the next user who requests the same page. You can do this by serializing the results, storing them in the database or in a file, and then checking if a cached version exists before a new page is generated.

The next example demonstrates how to build a class that can cache an array of values between requests. The class will work on files, but you can easily change it to store the values in a database. The caching class is stored in an include file (`cache.inc`) so it can be used in many different applications.

Table 10-7 lists the methods.

Table 10-7. *Caching Class Methods*

Name	Description
`__construct()`	Class constructor. Initiates properties.
`Check()`	Checks if the cache file exists and if it is still valid.
`Save()`	Writes the cached value to the file.
`SetValue()`	Adds or updates a value in the cache.
`GetValue()`	Retrieves a value in the cache.

The Code

```
<?php
// Example cache.inc
class Cache {
  private $name = null;
  private $value = array();
  private $ttl;

  function __construct($name, $ttl = 3600) {
    $this->name = $name;
    $this->ttl = $ttl;
  }

  function Check() {
    $cached = false;
    $file_name = $this->name . ".cache";
    if (file_exists($file_name)) {
      $modified = filemtime($file_name);
      if (time() - $this->ttl < $modified) {
        $fp = fopen($file_name, "rt");
        if ($fp) {
          $temp_value = fread($fp, filesize($file_name));
          fclose($fp);
          $this->value = unserialize($temp_value);
          $cached = true;
        }
      }
    }
    return $cached;
  }

  function Save() {
    $file_name = $this->name . ".cache";
    $fp = fopen($file_name, "wt");
    if ($fp) {
      fwrite($fp, serialize($this->value));
      fclose($fp);
    }
  }

  function SetValue($key, $value) {
    $this->value[$key] = $value;
  }
```

```
  function GetValue($key) {
    if (isset($this->value[$key])) {
      return $this->value[$key];
    }
    else {
      return NULL;
    }
  }
}
?>
```

How It Works

This caching class is used in the next example, where a cache object is created from the caching class and checked to see if the file exists. If not, the values are calculated and stored in the cache. If cached data exists and it is valid, the data will be retrieved and displayed.

The Code

```
<?php
// Example 10-8-2.php
include 'cache.inc';

$cache = new Cache('data');
if ($cache->Check()) {
  echo "Retrieving values from cache\n";
  $arr = $cache->GetValue('arr');
  $fruits = $cache->GetValue('fruits');
  print_r($arr);
}
else {
  $arr = array("apple", "orange", "apricot");
  $fruits = sizeof($arr);
  $cache->SetValue('arr', $arr);
  $cache->SetValue('fruits', $fruits);
  $cache->Save();
  echo "Values are stored in cache\n";
}
?>
```

How It Works

The first time the script is executed, the output will look like this:

```
Values are stored in cache
```

The second time, when the values are retrieved from the cache, it will look like this:

```
Array
(
    [0] => apple
    [1] => orange
    [2] => apricot
)
```

10-9. Sharing Variables Between Processes

When a user is navigating through a web application, it is useful to store user- or session-specific data on the web server so it is easy to access each time a page is requested. This can be information about the user, user preferences, or data related to the application, such as data in a shopping chart. Each time the user requests a page that contains a call to the `session_start()` function, the server will start a new process (or reuse an idle), and the PHP engine will look for a session ID in the query string or cookie data. This will fetch the saved session data and build the `$_SESSION` array.

As mentioned in recipe 10-5, `$_SESSION` is a superglobal and can be accessed directly from any code segment. When a session is active, it is possible to retrieve, add, update, and delete values from the `$_SESSION` array. You do this like any other variable. The engine will automatically store the values of the array when the script ends, unless it was stopped with an error. The session data file will be locked to keep multiple processes from accessing (writing to) the same data at the same time. If you have scripts that take a long time to execute or you are loading multiple frames from the same server, it might optimize the application to use `session_write_close()` or `session_commit()` to close the session data file. After either of these commands are used, it is not possible to add new values to the `$_SESSION` array.

Shared memory is another way of sharing data between processes. This is used when the two processes are running at the same time and might be started by different clients. Shared memory will in most cases be faster than a shared file or a table in a database. To use shared memory in PHP, it must be compiled with the `-enable-shmop` parameter.

■ **Note** Using shared memory requires that the processes are persistent such as Apache modules, IIS ISAPI, or PHP-GTK applications.

The `shmop` extension implements six simple functions, as shown in Table 10-8.

Table 10-8. *shmop Functions*

Name	Description
`shmop_open()`	Opens or creates a memory block for sharing
`shmop_close()`	Closes a shared memory block
`shmop_delete()`	Deletes a shared memory block
`shmop_read()`	Reads data from a shared memory block
`shmop_write()`	Writes data to a shared memory block
`shmop_size()`	Gets the size of a shared memory block

You must create a shared memory block before you can use it. You can use the `shmop_open()` function to do this; this function takes four arguments. The first is a unique ID (an integer) used to identify the memory block. The second parameter is a flag that specifies how the block is accessed (`a` = read-only, `c` = create or read/write, `w` = write and read, and `n` = create new or fail). The third argument specifies the access to the memory block and should be passed as an octal such as file system rights (for example, 0644). The fourth and last argument sets the size of the block. The third and fourth arguments should be set to 0 if you are opening an existing block.

■ **Note** The size of a shared memory block is fixed on creation and cannot be changed.

The following example shows how to create and write to a memory block. The block is deleted and closed at the end of the script, so in order to demonstrate how it works, the script will wait 60 seconds before it terminates. This should be enough time to run the next example and see the shared memory in action.

The Code

```
<?php
// Example 10-9-1.php
if (!extension_loaded("shmop")) {
  dl("php_shmop.dll");
}

$shm_id = shmop_open(0x123, 'c', 0644, 250);
shmop_write($shm_id, "Data in shared memory", 0);
$value = shmop_read($shm_id, 8, 6);
echo "$value";
shmop_delete($shm_id);
shmop_close($shm_id);
sleep(60);
?>
```

If the memory block should be used by another process, it should not be deleted, and another process could access the data, like this:

```
<?php
// Example 10-9-2.php
if (!extension_loaded("shmop")) {
  dl("php_shmop.dll");
}

$shm_id = shmop_open(0x123, 'a', 0, 0);
if ($shm_id) {
  $value = shmop_read($shm_id, 0, 100);
  echo "$value";
  shmop_close($shm_id);
}
?>
```

How It Works

Sharing memory between two scripts requires that both scripts run at the same time. The first script defines a shared memory block with a string and reads six bytes from the block. The second script connects to the same block through the same handle (0×123). The entire string is read and sent to the client.

10-10. Debugging

Printing and storing information during development and testing will help eliminate errors caused by variables having other values than expected or by using the wrong variable names. PHP implements several functions that make debugging a lot easier (see Table 10-9).

Table 10-9. *Functions Used for Debugging*

Name	Description
`echo()`	Prints a simple variable or value
`print()`	Prints a simple variable or value
`printf()`	Prints a formatted string
`var_dump()`	Prints the type and content of a variable
`print_r()`	Recursively prints the content of an array or object
`debug_backtrace()`	Returns an array with the call stack and other values

The functions `echo()`, `print()`, and `printf()` generate normal output, so using these to produce debug output might be a bit confusing, but this is the way to generate any output.

Having a function called `debug_print()` will make it easy to use debugging information and to turn it on and off when needed. This function could be defined in an include file along with a constant `DEBUG` set to `true` or `false`.

The Code

```
<?php
// Example debug.inc
define('DEBUG', true); // set to false for disabling

function debug_print($var) {
  if (DEBUG) {
    switch (strtolower(substr(php_sapi_name(), 0, 3))) {
      case 'cli' :
      var_dump($var);
        break;
      default :
        print("<pre>");
      var_dump($var);
        print("</pre>");
        break;
    }
  }
}
?>
```

How It Works

When the `DEBUG` constant is set to `true`, the function will generate output; when it is set to `false`, the function will be silent. This is an easy way to turn debug information on and off. The `debug_print()` function calls the `php_sapi_name()` function to determine how the PHP script is executed. Depending on the process type, it will generate different output.

Defining the `debug_print()` function in the file `debug.inc` makes it possible to reuse the same function in many scripts with a simple include statement and one or more calls to the function.

The Code

```
<?php
// Example 10-10-1.php
include 'debug.inc';

$a = array('orange', 'apple');
debug_print($a);
?>
```

How It Works

The include file with the debug information is included in the top of the script and used to print the content of an array.

```
array(2) {
  [0]=>
  string(6) "orange"
  [1]=>
  string(5) "apple"
}
```

PHP implements a few so-called magic constants. These are not really constants, because they change value depending on where they are used (see Table 10-10).

Table 10-10. *Magic Constants*

Name	Description
__FILE__	Name of current file
__LINE__	Current line number
__FUNCTION__	Name of current function
__CLASS__	Name of current class
__METHOD__	Name of current method

You can modify the debug_print() function from the previous example to use __FILE__ and __LINE__ to print where the debug information originated.

The Code

```
<?php
// Example debug1.inc
define('DEBUG', true); // set to false for disabling

function debug_print($var, $file = __FILE__, $line = __LINE__) {
  if (DEBUG) {
  $where = "File = $file ($line)";
    switch (strtolower(substr(php_sapi_name(), 0, 3))) {
      case 'cli' :
        echo "$where\n";
        var_dump($var);
        break;
      default :
        echo "$where<br>";
        print("<pre>");
        var_dump($var);
        print("</pre>");
        break;
    }
  }
}
```

```
?>
<?php
// Example 10-10-2.php
include 'debug1.inc';

$a = array('orange', 'apple');
debug_print($a, __FILE__, __LINE__);
?>
```

How It Works

In this example, you add two parameters to the `debug_print()` function. As shown in the following output, the `debug_print()` function can produce two forms of output. The first call to the function uses the default values for `$file` and `$line`. This causes the system to insert the name of the include file and the line where the function is defined. In the second call, you use `__FILE__` and `__LINE_` as parameters to the function call, and these will be replaced with the filename and line number where the function was called.

```
File = /Samples/debug1.inc (5)
array(2) {
  [0]=>
  string(6) "orange"
  [1]=>
  string(5) "apple"
}
File = /Samples/10-10-2.php (7)
array(2) {
  [0]=>
  string(6) "orange"
  [1]=>
  string(5) "apple"
}
```

Note how the two magic constants are used as default values for `$file` and `$line` in the definition of the function. If one or both of these two arguments are omitted from the call, they will be replaced by values that indicate the include file and the line where the function is defined.

Summary

This chapter demonstrated the strengths of PHP when it comes to variables and data types. The loosely typed behavior of PHP makes it easy to work with, and there is little reason to spend time on memory cleanups, as the engine handles these when the scripts terminate.

We discussed how variables are handled from creation, and we discussed how to manipulate data, how to test for values and types, and how to use the more advanced features of variable variables and functions.

We also showed examples of using the `serialize()` and `unserialize()` functions to format data so the data can be shared between calls or stored in a database. Finally, we showed some examples of how data can be shared between processes that run simultaneously.

Looking Ahead

The next chapter will discuss how functions are created and used in PHP.