CHAPTER 7

Functions

his chapter examines how you define and use functions. It introduces the general concept of functions before discussing the specifics of how PHP functions work; it covers the following topics:

- Defining functions
- Understanding variable scope in functions
- Managing function parameters
  - Parameters by value or reference
  - Parameter default values
  - Variable-length parameter lists
- Using functions to return values
- Managing dynamic function calls
- Using recursive functions

All programs contain instructions to perform tasks. When sets of tasks are frequently used to perform an activity, they are grouped into units, which are known as functions or methods in most programming languages. PHP calls these units *functions.*

Functions should perform well-defined tasks. They should also hide the complexity of their tasks behind a prototype. A prototype includes a function name, a list of parameters, and a return data type. The prototype should let you see how the function can be used in your programs.

Function names should be short, declarative descriptions about what tasks they perform. The list of parameters, also known as a *signature,* is typically enclosed in parentheses and should use descriptive variable names that signal their purpose when possible. The parameters in a function signature are considered formal at definition and actual at run time. In strongly typed languages, the parameters impose positional and data type restrictions. Weakly typed languages, like C or PHP, typically impose only positional restrictions. In some programming languages, formal parameters can also designate whether run-time values are passed by value or by reference. The return type is a valid data type in the programming language or a void, which means nothing is returned by the function.

Having reviewed the general concept of functions, you will see how to define functions in PHP in the next section. PHP functions are the product of a loosely typed programming language. This lineage offers you some advantages and challenges that will be discussed throughout the balance of the chapter. While the six sections are written independently, they are positioned to support each other and should be read in sequence.

Defining Functions

PHP functions are defined by a prototype that includes a function name, a list of parameters, and a return data type. The basic prototype of a PHP function that takes no parameters and returns no values is

```
void function function_name();
```

Function names in PHP must start with an alphabetical character or an underscore and consist of only alphabetical characters, numbers, or underscores. Also, *function names are global in scope and case insensitive,* which means functions can only exist once in a page and its set of included PHP libraries.

The nature of global-scope functions differs from the convention in other popular programming languages. Whether a function is defined in a page, inside another function, or in its included libraries, it is available anywhere in the PHP program. This means you cannot define local functions inside functions with the same expectations of behavior. User-defined functions also cannot have the same name as any PHP built-in function.

PHP is a mixed-case language, which means *functions are case insensitive* while virtually everything else is not. This can be confusing at times if you forget the rule. Case-insensitive functions enable you to enter their names in lower-, mixed-, or uppercase at different times in the same program. They will always refer to a single global-scope prototype, and the convention is to consistently use the same case as the function prototype. The convention exists because it makes reading and remembering function names easier.

TIP
You should avoid using function names more than once because they can conflict or be confused by programmers.

Along with these behaviors, *functions do not support overloading.* Overloading is the practice of having multiple function signatures that are accessed according to the position and type of actual parameters that are passed to a function at run time. PHP does not support overloading because it provides variable-length parameter lists. The concept of variable-length parameter lists is covered later in the chapter, in the section "Managing Function Parameters."

You will now define a function named `my_function()` that takes no parameters and returns nothing. It implements the following prototype:

```
void function my_function();
```

The following `Function1.php` program will print to the web page the string "`Hello PHP World!`"

```
-- This is found in Function1.php on the enclosed CD.

<?php
  // Call the user-defined function.
  print_string();

  // Define a no-parameter function that returns nothing.
  function print_string()
  {
    print "Hello PHP World!";
  }
?>
```

You can see that the keyword `void` in the prototype is not actually used in the program. The `function` keyword begins the definition, which is followed by the function name, a formal

parameter list in parentheses, and a set of statements enclosed in curly braces. This function has no formal parameters and is also known as a no-argument function.

Function calls are made by using a function name followed by parentheses enclosing no parameter, one parameter, or a list of parameters separated by commas. When function calls are used as expressions in condition statements, they are not followed by a semicolon because they act in the scope of an `if` statement and are enclosed in parentheses. Functions are followed by a semicolon when they are statements. Parameters can be any type of PHP variable or numeric or string literal values. Parameters are by default copies of the value or variable used when calling the function.

You define a function that takes a single formal parameter of a string with the following prototype:

```
void function my_function(string argument);
```

While you should try to have more meaningful names for prototype variables, you can use something generic, like *argument* here. You should note that formal parameters are preceded by a $ symbol when defining function signatures.

The following `Function2.php` program demonstrates passing a string literal as the only actual parameter to the `print_string()` function:

```
-- This is found in Function2.php on the enclosed CD.

<?php
  // Call the user-defined function.
  print_string("Hello PHP World!");

  // Define a parameter function that returns nothing.
  function print_string($argument)
  {
    print $argument;
  }
?>
```

The call to the `print_string()` function sends a string literal that will be printed to the rendered web page. As you see, *there is no type specified in the definition of function parameters.* This is by design because formal parameters are validated only by position, not data type. You will explore more on this topic in the section "Managing Function Parameters" later in the chapter.

In some programming languages, function definitions must precede function calls. This type of behavior, called *forward referencing,* is determined by how the program is parsed. Many parsers perform a single read from top to bottom, which requires a function to be defined or forward-referenced before it can be called. The PHP parser reads each delimited tag component block to load a page and then any included files to build a complete program. This is done because parts of the programs can be located between discrete tags in the same page.

After reading the PHP code segments, function definitions are first read into memory, which defines them in the page. After reading functions into memory, the rest of the page is processed. This process avoids fatal undefined function run-time errors. This works well except when a function is nested in another function, because *nested functions are not defined in the global memory space until the enclosing function is called.* This creates the possibility that a program

may require forward referencing to run correctly. Large programs sharing a dependency on nested functions can be hard to debug and fix. Therefore, you should *avoid writing function* ***definitions*** *within functions.*

PHP recognizes the value of having anonymous, or lambda-style, functions because they allow you to modularize coding components without adding them to the global memory space. You define an anonymous function by using the following pattern:

```
string create_function (string args, string code)
```

The first string contains a list of formal parameters to the function, and the second contains the code for the anonymous program. You declare a variable by assigning it the predefined `create_function()` return value. Then, *you can treat the variable as a function and pass actual parameters to the variable in appended parentheses.*

The following `DynamicFunction1.php` program demonstrates this concept:

```
-- This is found in DynamicFunction1.php on the enclosed CD.

<?php
  // Define a set of variables.
  $a = 2;
  $b = 2;
  $c = 4;

  // Call the global function.
  nested($a,$c);

  // Call the anonymous function by its variable name.
  if (isset($add))
    print "[".$add($a,$b)."]";
  else
    print "[\$add is not defined.]";

  // Define a global function.
  function nested($a,$b)
  {
    // Define an anonymous function.
    $add = create_function('$a,$b','return $a + $b;');

    // Call the anonymous function by its variable name.
    if (isset($add))
      print "[".$add($a,$b)."]<br>";
  }
?>
```

The program will add the contents of variables `$a` and `$c` by passing them as actual parameters to the `nested()` function. The results produced are shown in the following output:

```
[6]
[$add is not defined.]
```

Anonymous functions defined within the scope of functions are truly local in scope because they exist only in the scope of the function. When you define anonymous functions at the PHP script level, the functions' scope is equal to that of the variable. You will need to use the `unset()` function to remove the variable if you want the anonymous function destroyed after execution.

You have learned how to define functions and avoid nested functions because of their runtime-loading dependency. The next section will explore how variable scope impacts functions.

Understanding Variable Scope in Functions

PHP functions work differently than many other programming languages because they are globally defined. Likewise, the rules of scope may appear different than the norm, but they are not.

Functions can access only global variables and variables defined in them, which are known as *local* variables. You define global variables with the `define()` function that was discussed in Chapter 4. Global variables can be defined inside functions, which make them available only after calling the containing function. This behavior has benefits and problems. The benefit is that you can define conditional global variables at run time. The problem is that you can have more than one mechanism that returns values from a function, which is a potential control problem.

You can also define another type of global variable by using the `GLOBAL` keyword in a function. *Creating a variable with the `GLOBAL` keyword does not create an environment global variable like one created with the `define()` function.* The `GLOBAL` keyword creates a variable that you can access externally from the function where it is defined. The global variable's scope is equal to variables defined at the PHP scripting tag level. This means we have two types of global variables that have different scope. Labeling these behaviors is difficult but you can view the *`GLOBAL` keyword as defining script-level global variables, whereas the `define()` function creates environment-level global variables.* The latter can be accessed inside and outside of functions, while the former cannot be accessed in other functions unless passed as an actual parameter.

The following `Function3.php` program demonstrates the access privilege of a function to read an environment variable and set global environment- and script-level variables:

```
-- This is found in Function3.php on the enclosed CD.

<?php
  // Declare a user-defined global variable.
  define('LOCAL',"[The global variable]");

  // Call function.
  print_string();

  // Print function declared local variable.
  print $function_local;

  // Define a function.
  function print_string()
  {
    // Define a global scoped variable.
    GLOBAL $function_local;
```

```
    // Print the local variable with a text literal.
    $function_local = LOCAL." is inside the function.";
  }
?>
```

The `Function3.php` program uses the `define()` function to build a global environment variable. The `print_string()` function *uses the `GLOBAL` keyword to build a script-level global variable.* You will notice that you must define a global variable on a separate line before assigning a value. The next line joins a global environment-level variable and a string literal. The program shows you can access an environment variable and define a script-level global variable inside a function. Then, you can access the variable outside of the function where it was defined. It prints the following output to a page:

```
[The global variable] is inside the function.
```

If you add the following additional function at the end of the `Function3.php` program, it will demonstrate the scope limit of script-level global variables:

```
// Define a function to fail on scope.
function script_global()
{
  print "[\$function_local][".$function_local."]<br>";
}

// Call the failing function.
script_global();
```

Adding the preceding code to the `Function3.php` program will raise an undefined variable exception in the `script_global()` function. This happens because *the subsequent function lacks scope to access the `$function_local` script-level variable.*

When you define a script-level variable without using the `GLOBAL` keyword and try to access it in a function, the same scope error happens. This occurs because a variable is defined as external to the scope of the function. You must pass the variable as an actual parameter to access it inside a function. The following `Function4.php` program demonstrates this:

```
-- This is found in Function4.php on the enclosed CD.

<?php
  // Declare a variable.
  $local_var = "Local variable";

  // Define a function that prints a string.
  function print_string()
  {
    // Print the local variable with a text literal.
    print "[".$local_var."]";
  }

  // Print string.
  print_string();
?>
```

The `Function4.php` program *will raise a nonfatal notice* if you have enabled the `display_errors` directive in the `php.ini` file, *or it will return empty brackets ([]) in the web page otherwise.* The program demonstrates that with or without a `GLOBAL` keyword, script-level variables are out of scope to the internal operation of functions, unless they are passed as actual parameters. Only global variables built by using the `define()` function are in scope to functions without explicitly passing them as actual parameters.

The same scope rules apply to the anonymous, or lambda-style, functions discussed earlier in the chapter. When an anonymous function is stored in a target variable that is defined as global within a function, the variable follows the same rules as ordinary script-level variables. You can see this principle demonstrated in the `DynamicFunction2.php` program:

```
-- This is found in DynamicFunction2.php on the enclosed CD.

<?php
  // Define a set of variables.
  $a = 2;
  $b = 2;
  $c = 4;

  // Call the global function.
  nested($a,$c);

  // Call the anonymous function by its variable name.
  if (isset($add))


    print "[".$add($a,$b)."]";
  else
    print "[\$add is not defined.]";

  // Define a global function.
  function nested($a,$b)
  {
    // Define a script-level variable.
    GLOBAL $add;

    // Define an anonymous function for a script-level variable.
    $add = create_function('$a,$b','return $a + $b;');

    // Call the anonymous function by its variable name.
    if (isset($add))
      print "[".$add($a,$b)."]<br>";
  }
?>
```

The program defines a script-level `$add` variable internally in the `nested()` function. The variable `$add` is accessible as a function externally to where it is defined. You can redefine or delete the behavior of the function after its definition. The call to the anonymous function outside

of the `nested()` function uses the script-level values of `$a` and `$b`, which sum to 4, as shown in the following output:

```
[6]
[4]
```

You have learned how variable scope impacts functions whether they contain ordinary data types or anonymous functions. The next section will explore how to manage the definition and use of parameters.

Managing Function Parameters

In this section, you will learn how to use and manage parameters to build effective functions. While functions that don't use parameters typically do very little, they can be very useful for fixed or static tasks. Parameterized functions provide dynamic code blocks that meet more complex programming tasks often required by business rules.

Parameters by Value or Reference

As discussed in the introduction to this chapter, parameters are passed by value or reference to functions. Passing a parameter by value means that you hand a copy of the variable or literal value to a function. The alternative to passing a parameter by value is to pass one by reference, which means that you provide a reference or pointer to the variable in memory. The former ensures that the original variable is unchanged by the function activity, while the latter enables the function to change the original variable value.

The `FunctionByValue.php` program demonstrates how passing by value leaves the actual parameter value unchanged:

```
-- This is found in FunctionByValue.php on the enclosed CD.

<?php
  // Declare variable.
  $var = "Initial";

  // Print actual parameter value before function.
  print "Before function \$var [".$var."]<br>";

  // Call the user-defined function.
  print_string($var);

  // Print actual parameter value after function.
  print "After function \$var [".$var."]";

  // Define function that prints an input parameter as a string.
  function print_string($var)
  {
    // Print the local variable with a text literal.
    $var = "Updated";
```

```
    // Print actual parameter value.
    print "Inside function \$var [".$var."]<br>";
  }
?>
```

The program uses a single `$var` variable name, but the variable is actually two variables that share the same name but have different scope. The declaration of `$var` as a string sets the value as "Initial" for a script-level global variable. The second declaration is subtle because it occurs in the definition of the `print_string($var)` function. The formal parameter `$var` has a local scope limited to the function and is passed by value, which is the default. The names are the same, but their scopes are different. Inside the function the local `$var` variable is redefined as a string literal of "Updated" and printed.

The output from the `FunctionByValue.php` program demonstrates that the value passed by the call to the `print_string()` function was a copy of a script-level global variable because the variable value before and after the function remains the same. The following output is produced:

```
Before function $var [Initial]
Inside function $var [Updated]
After function $var [Initial]
```

The benefit of passing by value is that the copy of a variable can be used to produce something of value to the program without altering the original variable. Passing by value is the default behavior in PHP because functions act like black boxes that take input and return a derived output.

It is possible that you want the function to return a new value to replace the old value of the actual parameter variable. You can do so by returning the modified value from the function. Then, you assign the function return value to the same variable passed as the actual parameter to the function. Assuming you now return a value from the `print_string($var)` function, you can use the following syntax:

```
$var = print_string($var);
```

This solution works when there is only one variable entering and exiting a function. When you have two or more formal parameters and you want one or both changed, you will need to pass by reference.

The `FunctionByReference.php` program demonstrates how passing by reference changes the actual parameter value:

```
-- This is found in FunctionByReference.php on the enclosed CD.

<?php
  // Declare variable.
  $var = "Initial";

  // Print actual parameter value before function.
  print "Before function \$var [".$var."]<br>";

  // Call the user-defined function.
  print_string($var);
```

```
  // Print actual parameter value after function.
  print "After function \$var [".$var."]";

  // Define function that prints an input parameter as a string.
  function print_string(&$var)
  {
    // Print the local variable with a text literal.
    $var = "Updated";

    // Print actual parameter value.
    print "Inside function \$var [".$var."]<br>";
  }
?>
```

The key difference between `FunctionByReference.php` and the prior program is that here, the function *defines the formal parameter as a reference. This is done by placing an ampersand before the variable name.* The ampersand instructs the PHP compiler to pass a reference, not a copy, to the function.

The following output demonstrates that the original variable is altered inside the function:

```
Before function $var [Initial]
Inside function $var [Updated]
After function $var [Updated]
```

TIP

The `FunctionByReference.php` program uses `$var` as the formal parameter name to illustrate that one variable name can have two or more scopes. You can and should typically rename the `$var` formal parameter in the `print_string()` function to avoid confusion.

NOTE

Prior to PHP 5, you could pass a reference into a function, but that behavior has been deprecated. Now you define the function to process parameters by reference.

You have now learned how to pass parameters by value or by reference in PHP. Choosing when to do so is the trick. There are valid reasons to support both approaches, but you should consider carefully when you choose to pass by reference, because it is a form of coupling, which can add unnecessary complexity to your PHP programming library.

Parameter Default Values

When defining functions, you can define parameters as mandatory or optional. *Mandatory* parameters must be listed before optional parameters in the argument list. *Optional* parameters have a default value, which can be any scalar variable, literal, or null value.

The most generic prototype to demonstrate this behavior is

```
void function add_numbers(int a = 0, int b = 0);
```

The following `DefaultValue.php` program demonstrates default values by making both formal parameters optional and their default values zero:

```
-- This is found in DefaultValue.php on the enclosed CD.

<?php
  // Call function without actual parameters.
  add_numbers();

  // Call function with two actual parameters.
  add_numbers(2,2);

  // Define a function that adds two numbers.
  function add_numbers($a = 0,$b = 0)
  {
    // Add two numbers and print the result.
    print "[(\$a + \$b)][".($a + $b)."]<br>";
  }
?>
```

The program demonstrates a call to the `add_numbers()` function without actual parameters and a call with two actual parameters. The first call will use the default zero values and results in a sum of zero, while the second call will ignore the default values and add the numbers provided in the function call. The `DefaultValue.php` program generates the following output:

```
[($a + $b)][0]
[($a + $b)][4]
```

Using default values in this example is a good fix to a simple problem. On the other hand, operational parameters pose other programming challenges. The sequential ordering dependency of optional variables requires all variables be provided from left to right. When one variable in a sequence of variables is skipped, a function can provide an incorrect result. Likewise, the absence of actual parameter run-time type identification can cause abnormal results by performing unexpected and implicit type casting operations.

Both of these problems exist because PHP uses variable-length parameter lists for function signatures. You will explore how to leverage variable-length parameter lists in the next section.

Variable-Length Parameter Lists

Variable-length parameter lists are common patterns in programming languages. The C, C++, C#, and Java programming languages all support variable-length parameter lists, but they label them differently. A *variable-length* parameter list is an array or a list of values, where the values are valid PHP data types.

When you learned earlier in the chapter how to build and implement a prototype, you had two options. One option was to use parameters, and the other was not to use them. As discussed earlier, there are two parameter options: mandatory or optional. These options make PHP function parameter lists more complex.

A function definition or prototype that uses a single mandatory parameter requires that you call the function with at least one actual parameter but does not restrict you from passing more than one. You can actually submit any number of parameters beyond the mandatory number

Function	Description and Pattern
`func_get_arg()`	The function takes one formal parameter, which is the index value in the variable-length parameter list. When the actual parameter is found in the range of the parameter list indexes, the function returns that argument value. If the index value is not found in the list, the function raises a warning and returns a null value. The function has the following pattern: `mixed func_get_arg(int arg_num)`
`func_get_args()`	The function takes no formal parameters and returns a numerically indexed array of arguments. If there are no parameters passed to the function, a null array is returned. The null array has zero elements, and attempting to access element zero will raise a nonfatal error. It has the following pattern: `array func_get_args()`
`func_num_args()`	The function takes no formal parameters and returns the number of elements in the argument list. The valid range is from 0 to the maximum number of parameters. The function has the following pattern: `int func_num_args()`

TABLE 7-1. *Variable-Length Parameter List Functions*

required by a function prototype. You can define functions without any parameters and still manage a parameter list passed to the function, which means *prototypes are optional.*

After introducing some enabling functions, you will see how to manage variable-length parameter lists in PHP functions. Table 7-1 qualifies predefined functions that enable flexible parameter list management.

The *variable-length parameter list functions can* ***only*** *be used inside of a function definition and have no valid context outside of functions.* Any attempt to call these outside of the scope of a function can raise a nonfatal warning message.

The following `FlexibleParameters.php` program demonstrates the use of these functions:

```
-- This is found in FlexibleParameters.php on the enclosed CD.

<?php
  // Call function.
  print_string("Oh yeah, baby!","There's no prototype required.");

  // Define a function that prints a string.
  function print_string()
  {
    // Check for actual parameters.
    if (func_num_args() > 0 )
```

```
      // Process list of actual parameters.
      foreach (func_get_args() as $index => $arg)
        print "Parameter List [".$index."][".$arg."]<br>";
  }
?>
```

The program *demonstrates two actual parameters passed in the call* to the `print_string()` function, though the prototype indicates that *there are no formal parameters.* This may look quirky but it is actually a natural feature of flexible parameter passing. When you call a function, PHP checks the list of actual parameters to ensure that there are at least enough elements to match the number defined by the function prototype. Additional values are not validated because they are assumed to be optional parameters defined by the prototype or superfluous and ignored.

The `FlexibleParameters.php` program will display the following to a web page:

```
Parameter List [0][Oh yeah, baby!]
Parameter List [1][There's no prototype required.]
```

You have now learned how to access all actual parameters passed to user-defined functions whether defined in the prototype or not. This also completes the discussion on how you manage function parameters.

In the next section, you will examine how variable-length parameter lists and type validation functions can evaluate and process function parameters to guarantee correct return results.

Using Functions to Return Values

Up to this point, you have experimented with user-defined functions that do not return values. The traditional black-box value of functions is that inputs are processed into meaningful outputs. The outputs are return values from functions. The broadest prototype of a function that returns any data type is

```
mixed function my_function(mixed var);
```

This prototype basically says that the function has a single input of a variable that may be any data type and returns a variable of any data type. Unfortunately, many functions work accidentally this way because of careless type management of return values.

Function return types are provided by using the `return` keyword, a variable or literal value, and a semicolon. You can return any valid data type from functions. Unlike in many other programming languages, you can treat a PHP function that returns a variable like one that does not, simply ignoring the return rather than assigning it to a left operand.

The following prototype will be used to build a small calculator function:

```
float function calculate(int a = 0, int b = 0, int op = 0, array operand);
```

The function has four optional formal parameters: the first three are integers and the last is an array of operand values. The default values speak for themselves. As discussed in the preceding section, there is an immediate problem with the prototype because there is no way to enforce the ordering or provisioning of the actual parameters.

The solution to managing these optional parameters lies in effective use of the variable-length parameter list functions combined with the type identification functions. The list of type identification functions is found in Table 7-2.

Function	Description and Pattern
`is_array()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is an array and false if it is anything else. The function has the following pattern: `bool is_array(mixed var)`
`is_bool()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is a Boolean and false if it is anything else. The function has the following pattern: `bool is_bool(mixed var)`
`is_float()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is a float and false if it is anything else. The `is_double()` and `is_real()` functions are aliases to the `is_float()` function. The function has the following pattern: `bool is_float(mixed var)`
`is_int()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is an integer and false if it is anything else. The `is_integer()` and `is_long()` functions are aliases to the `is_int()` function. The function has the following pattern: `bool is_int(mixed var)`
`is_null()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is a null and false if it is anything else. The function has the following pattern: `bool is_null(mixed var)`
`is_numeric()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is a number or a numeric string and false if it is anything else. The function has the following pattern: `bool is_numeric(mixed var)`
`is_object()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is an object and false if it is anything else. The function has the following pattern: `bool is_object(mixed var)`
`is_scalar()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is a Boolean, float, integer, or string, and false if it is anything else. The function has the following pattern: `bool is_scalar(mixed var)`
`is_string()`	The function takes one formal parameter, which can be any data type. The function returns true if the actual parameter is a string and false if it is anything else. The function has the following pattern: `bool is_string(mixed var)`

TABLE 7-2. *Type Identification Functions*

The `Calculate.php` program demonstrates how to leverage variable-parameter lists and data type functions together to solve a problem. The program follows:

```
-- This is found in Calculate.php on the enclosed CD.

<?php
  // Define a global variable.
  GLOBAL $operands;

  // Assign the global variable an array of operands.
  $operands = array("+","-","*","/");

  // Call function with each operand.
  for ($i = 0;$i < count($operands);$i++)
  {
    // Print call and return value.
    print "[(\$a = 86, \$b = 6, \$operand = ".$operands[$i].")]";
    print "[".calculate(86,(6 + $i),$i,$operands)."]<br>";
  }

  // Define a function that does binary math operations.
  function calculate($a = 0, $b = 0,$op = 0,$operands)
  {
    // Check for three actual parameters.
    if (func_num_args() == 4)
    {
      // Check and override the default value.
      if (is_numeric(func_get_arg(0)))
        $a = func_get_arg(0);
      else
        return null;

      // Check and override the default value.
      if (is_numeric(func_get_arg(1)))
        $b = func_get_arg(1);
      else
        return null;

      // Check and override the default value and choose operation.
      if ((is_numeric(func_get_arg(2))) &&
          (array_key_exists(func_get_arg(2),$operands)))
      {
          // Declare operand.
          $op = $operands[func_get_arg(2)];

          // Apply formula based on operand.
          switch (true)
```

```
            {
              case ($op == "+"):
                return (float) $a + $b;
              case ($op == "-"):
                return (float) $a - $b;
              case ($op == "*"):
                return (float) $a * $b;
              case ($op == "/"):
                return round((float) $a / $b,2);
            }
        }
        else
        {
          // Return a null argument error.
          return null;
        }
      }
      else
      {
        // Return a null argument error.
        return null;
      }
    }
?>
```

The `Calculate.php` program checks if only four actual parameters are passed at run time before evaluating any arguments. If there are four actual parameters, the program validates whether the first two are numeric values and assigns the numeric values to the first two actual parameters. The program evaluates the third to see if it is a number and then uses the number to determine if a value is found in the fourth actual parameter, which is the `$operands` array. A match in the `$operands` array is assigned to the third formal parameter, which implicitly casts it to a string. *If anything doesn't meet the appropriate test, it is discarded and a null value is returned by the function.*

The return values are explicitly typecast to floats for two reasons. One is that it is possible that a real number can be returned as a division product; and the other is that all numbers should be returned as the same data type. The `case` statement selection of an operator returns the mathematical result to the calling program unit. The following output is generated by the program:

```
[($a = 86, $b = 6, $op = +)][92]
[($a = 86, $b = 6, $op = -)][79]
[($a = 86, $b = 6, $op = *)][688]
[($a = 86, $b = 6, $op = /)][9.56]
```

This technique demonstrates leveraging a variable-length parameter list and type identification functions to guarantee position and type signature behaviors. Together, variable-length parameter lists and type identification functions enable you to validate function signatures and demonstrate how to return values from functions. The last section in this chapter will explore how to build recursive functions.

Managing Dynamic Function Calls

Chapter 6 shows you how to pass a customized sort function into a predefined sorting function. The ability to do so, in your own library functions, is a powerful tool. This section will demonstrate how you pass a function name to make a dynamic function call.

The mechanism to make dynamic function calls is tied to how variables are managed in PHP. You can define a function in your page, or library, and then pass the name of the function as a string and its parameters as a list to another function. Inside the function, you append parentheses containing the actual parameter list to the variable that contains the function name. This statement makes a dynamic function call to the function represented by the string and uses a parameter list that can contain static values, variables, or an array of values.

As covered in Chapter 6, the `usort()` predefined function takes an array by reference and a callback function. Internally, the `usort()` function manages how values from the array are passed into your user-defined callback function. The function also builds a new array by replacing the old array with a sorted one.

The `DynamicFunctionCall.php` program demonstrates a `user_sort()` function that takes two parameters and has the following prototype:

```
void user_sort(string function, array var)
```

You provide the name of the function as the first parameter and the target array as the second parameter. The array is passed by reference, which means that the array will be sorted inside the function. The `DynamicFunctionCall.php` program defines the `user_sort()` function to demonstrate dynamic function calls. *The program builds a dynamic run-time function call by concatenating a function name variable and parentheses containing the array variable.*

Three supporting functions are provided in the program. Two are classic sorting algorithms—the bubble and cocktail sorts. The third is a `switch()` library function that you may find very useful. The `switch()` function enables you to change the elements in an array to be by reference, a capability that isn't provided in the predefined array management library.

TIP

The bubble and cocktail sort programs work only with numeric arrays and will fail if you attempt to use an associative array when the keys are unique strings as opposed to numbers.

The following `DynamicFunctionCall.php` program demonstrates how you pass a function by reference to call dynamically at run time:

```
-- This is found in DynamicFunctionCall.php on the enclosed CD.

<?php
  // Declare sample arrays.
  $array1 = array(7,3,2,1,6,5,4);
  $array2 = array(3,7,2,6,5,4,8);

  // Choose a bubble sort.
  user_sort('bubble_sort',$array1);
```

```
// Choose a cocktail sort.
user_sort('cocktail_sort',$array2);

// Print the bubble sort results.
print "[".print_r($array1,true)."]<br>";

// Print the cocktail sort results.
print "[".print_r($array2,true)."]<br>";

// Performs a function by reference.
function user_sort($function,&$array)
{
  // Build function statement at runtime.
   $function($array);
}

// Performs a classic bubble sort passing array by reference.
function bubble_sort(&$array)
{
  // Read contents with one index.
  for ($i = 0;$i < count($array);$i++)
  {
    // Read contents with another index.
    for ($j = 0;$j < count($array);$j++)
    {
      // Check outer less than inner or switch by reference.
      if ($array[$i] < $array[$j])
        swap($array[$i],$array[$j]);
    }
  }
}

// Performs a classic cocktail sort passing array by reference.
function cocktail_sort(&$array)
{
  // Declare logical control variable.
  $done = false;

  // Declare boundary variables.
  $left = 0;
  $right = count($array);

  do
  {
    // Set exit criteria.
    $done = true;

    // Decrement right boundary to avoid index overrun.
    --$right;
```

```
      // Read from left to right boundary.
      for ($i = $left;$i < $right;$i++)
      {
        // Check current greater than next and switch by reference.
        if ($array[$i] > $array[$i+1])
          swap($array[$i],$array[$i+1]);
          $done = false;
      }

      // Read from right to left boundary.
      for ($i = $right;$i > $left;$i--)
      {
        // Check current less than next and switch by reference.
        if ($array[$i] < $array[$i-1])
          swap($array[$i],$array[$i-1]);
          $done = false;
      }

      // Increment left boundary.
      $left++;

    } while (!$done);
  }

  // Swap two variables by reference.
  function swap(&$a,&$b)
  {
    $c = $a;
    $a = $b;
    $b = $c;
  }
?>
```

The output generated from the `DynamicFunctionCall.php` program is

```
[Array ( [0] => 1 [1] => 2 [2] => 3 [3] => 4 [4] => 5 [5] => 6 [6] => 7 ) ]
[Array ( [0] => 2 [1] => 3 [2] => 4 [3] => 5 [4] => 6 [5] => 7 [6] => 8 ) ]
```

This is a powerful tool, but there is one caveat to using it. When you call a function dynamically, the logic should ensure that all functions have the same signature. Having the same signature means that they have the same count of parameters. Alternatively, you can easily pass a single parameter, provided it is an array that contains the list of arguments. This approach leverages variable-length parameter lists covered earlier in the chapter.

Using Recursive Functions

The ability of a function to call a copy of itself is called *recursion.* Recursion is useful to solve many types of programming problems, such as parsing and node tree searches. There are two types of recursion: linear and nonlinear. In linear recursion, a function calls only one copy of itself each time. In nonlinear recursion, a function calls more than one copy of itself each time.

The only problem with recursion is that it consumes large amounts of system resources. For example, PHP *programs that perform over 200 recursive calls run the risk of collapsing the PHP stack,* which would mean a depth of 200 linear recursions. Nonlinear recursions require more memory and will probably collapse the PHP stack in half or less the depth of a linear recursion.

The following `Recursion.php` program demonstrates linear recursion by using the classic factorial problem:

```
-- This is found in Recursion.php on the enclosed CD.

<?php
  // Call the recursive program.
  print "[factorial(7)] = [".factorial(7)."]<br>";

  // Define a recursive function that returns a double.
  function factorial($var)
  {
    // Check number of actual parameters.
    if (count($argv = func_get_args()) == 1);
    {
      if ($argv[0] <= 1)
        return (double) 1;
      else
        return (double) $argv[0] * factorial($argv[0] - 1);
    }
  }
?>
```

The program calls the `factorial()` function; and the `factorial()` function validates the argument list before proceeding with the calculation. If the actual parameter is not less than or equal to 1, the function calls one more copy of itself. The nested call will check the same exit condition and call itself again until it meets the exit condition of being 1 or less. The `Recursion.php` program will call itself six times before meeting the exit condition. Once the exit condition is met, it will start returning values from the deepest level to the original external call to the function.

It is unlikely that you need to use recursion to resolve mathematical problems. On the other hand, you will need to format many HTML tables. Some tables can contain one or more nested tables, and the nested tables can occur in different rows or columns. While you could write a number of programs to manage the different scenarios, writing one recursive function can make your life easier.

The following `RecursiveArray.php` program assumes a single format for each table to render a slightly modified multidimensional array from Chapter 6:

```
-- This is found in RecursiveArray.php on the enclosed CD.

<?php
  // Declare an asymmetrical multidimensional array.
  $tolkien = array
             ("Bilbo"=>array
                ("Hobbit"=>"Who slew a dragon for dwarves."
                ,"Fellowship"=>"Left the ring for Frodo."
```

```
                ,"Return of the King"=>array
                    ("Action 1"=>"Destroyed the one ring."
                    ,"Action 2"=>"Left middle earth."))
             ,"Frodo"=>array
                ("Fellowship"=>"Who inherited the ring."
                ,"Return of the King"=>"Who destroyed the ring.")
             ,"Gildor"=>"A high elf met on the road."
             ,"Sam"=>array
                ("Fellowship"=>"An eavesdropper at the window."
                ,"Two Towers"=>"A lone companion of Frodo."
                ,"Return of the King"=>"A ring bearer too.")
             );


  // Print the table and nested tables.
  print nested_tables($tolkien);

  // A recursive program for rendering nested tables.
  function nested_tables($array_in)
  {
    // Declare variable with table start.
    $output = "<table border=0 cellpadding=0 cellspacing=0>";

    // Read and add array keys and values to a string.
    foreach ($array_in as $hashName => $hashValue)
    {
      // Append row tag and first column value.
      $output .= "<tr><td valign=top>[".$hashName."]</td>";

      // Check column value for nested array.
      if (is_array($hashValue))
      {
        // Call recursively a copy of itself with cell tags.
        $output .= "<td>".nested_tables($hashValue)."</td></tr>";
      }
      else
      {
        // Append non-array value with cell tags.
        $output .= "<td>[".$hashValue."]</td></tr>";
      }
    }

    // Close HTML table.
    $output .= "</table>";

    // Return the HTML table.
    return $output;
  }
?>
```

The program renders an HTML table when you submit a single-dimensional or multidimensional array; and multidimensional arrays can be asymmetrical or symmetrical. The `RecursiveArray.php` program produces the following output for the asymmetrical multidimensional array:

[Bilbo]	[Hobbit]	[Who slew a dragon for dwarves.]
	[Fellowship]	[Left the ring for Frodo.]
	[Return of the King]	[Action 1] [Destroyed the one ring.]
		[Action 2] [Left middle earth.]
[Frodo]	[Fellowship]	[Who inherited the ring.]
	[Return of the King]	[Who destroyed the ring.]
[Gildor]	[A high elf met on the road.]	
[Sam]	[Fellowship]	[An eavesdropper at the window.]
	[Two Towers]	[A lone companion of Frodo.]
	[Return of the King]	[A ring bearer too.]

The `nested_tables()` function provides a single set of code that enables you to process various scenarios of nested tables or ordinary single-dimensional tables. The recursive test is a little bit trickier to see, but it is done by the `if` statement that checks if a hash value is an array.

TIP
Where possible, you should look for recursive solutions because they can simplify your code.

NOTE
Make sure when you call recursively that you pass the hash value, not the original array variable, which would cause an infinite recursive loop that will crash your Apache server.

You have covered recursive functions and learned to be cautious of how deep your recursions may go before implementing them in your solution set. The collapse of the stack is a fatal error.

Summary

You have learned how to define, manage, and use functions. These skills will enable you to build modular code where appropriate and position you to understand object technology in the next chapter.