Chapter 3. Functions

Every computer program in every language is built by tying various components of business logic together. Often, these components need to be somewhat reusable, encapsulating common functionality that needs to be referenced in multiple places throughout an application. The easiest way to make these components modular and reusable is to encapsulate their business logic into *functions*, specific constructs within the application that can be referenced elsewhere throughout an application.

Example 3-1 illustrates how a simple program might be written to capitalize the first character in a string. Coding without using functions is considered *imperative* programming, as you define exactly what the program needs to accomplish one command (or line of code) at a time.

Example 3-1. Imperative (function-free) string capitalization

```php
$str = "this is an example";

if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {
    $str[0] = chr(ord($str[0]) - 32);
}

echo $str . PHP_EOL; // This is an example

$str = "and this is another";

if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {
    $str[0] = chr(ord($str[0]) - 32);
}

echo $str . PHP_EOL; // And this is another

$str = "3 examples in total";

if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {
    $str[0] = chr(ord($str[0]) - 32);
}
```

```
echo $str . PHP_EOL; // 3 examples in total
```

NOTE

The functions `ord()` and `chr()` are references to native functions defined by PHP itself. The `ord()` function returns the binary value of a character as an integer. Similarly, `chr()` converts a binary value (represented as an integer) into its corresponding character.

When you write code without defining functions, your code ends up rather repetitive as you're forced to copy and paste identical blocks throughout the application. This violates one of the key principles of software development: *DRY*, or *don't repeat yourself*.

A common way to describe the *opposite* of this principle is *WET*, or *write everything twice*. Writing the same block of code over again leads to two problems:

- Your code becomes rather long and difficult to maintain.
- If the logic within the repeated code block needs to change, you have to update *several* parts of your program every time.

Rather than repeating logic imperatively, as in Example 3-1, you can define a function that wraps this logic and later invoke that function directly, as in Example 3-2. Defining functions is an evolution of imperative to procedural programming that augments the functions provided by the language itself with those defined by your application.

Example 3-2. Procedural string capitalization

```
function capitalize_string($str)
{
    if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {
        $str[0] = chr(ord($str[0]) - 32);
    }

    return $str;
}
```

```
$str = "this is an example";

echo capitalize_string($str) . PHP_EOL; // This is an e

$str = "and this is another";

echo capitalize_string($str) . PHP_EOL; // And this is

$str = "3 examples in total";

echo capitalize_string($str) . PHP_EOL; // 3 examples i
```

User-defined functions are incredibly powerful and quite flexible. The `capitalize_string()` function in Example 3-2 is relatively simple—it takes a single string parameter and returns a string. However, there is no indication in the function as defined that the `$str` parameter must be a string—you could just as easily pass a number or even an array as follows:

```
$out = capitalize_string(25); // 25

$out = capitalize_string(['a', 'b']); // ['A', 'b']
```

Recall the discussion of PHP’s loose type system from Chapter 1--by default, PHP will try to infer your intent when you pass a parameter into `capitalize_string()` and, in most cases, will return something useful. In the case of passing an integer, PHP will trigger a warning that you are trying to access elements of an array incorrectly, but it will still return an integer without crashing.

More sophisticated programs can add explicit typing information to both the function parameters and its return to provide safety checks around this kind of usage. Other functions could return *multiple* values rather than a single item. Strong typing is illustrated explicitly in Recipe 3.4.

The recipes that follow cover a variety of ways functions can be used in PHP and begins scratching at the surface of building a full application.

3.1 Accessing Function Parameters

Problem

You want to access the values passed into a function when it’s called elsewhere in a program.

Solution

Use the variables defined in the function signature within the body of the function itself as follows:

```
function multiply($first, $second)
{
    return $first * $second;
}

multiply(5, 2); // 10

$one = 7;
$two = 5;

multiply($one, $two); // 35
```

Discussion

The variable names defined in the function signature are available only within the scope of the function itself and will contain values matching the data passed into the function when it’s called. Inside the curly braces that define the function, you can use these variables as if you’ve defined them yourself. Just know that any changes you make to those variables will *only* be available within the function and won’t impact anything elsewhere in the application by default.

Example 3-3 illustrates how a specific variable name can be used both within a function and outside a function while referring to two, completely independent values. Said another way, changing the value of `$number` within the function will only impact the value within the function, not within the parent application.

Example 3-3. Local function scoping

```
function increment($number)
{
    $number += 1;

    return $number;
}

$number = 6;

echo increment($number); // 7
echo $number; // 6
```

By default, PHP passes values into functions rather than passing a reference to the variable. In Example 3-3, this means PHP passes the *value* `6` into a new `$number` variable within the function, performs a calculation, and returns the result. The `$number` variable outside the function is entirely unaffected.

WARNING

PHP passes simple values (strings, integers, Booleans, arrays) by value by default. More complex objects, however, are *always* passed by reference. In the case of objects, the variable inside the function points back to the same object as the variable outside the function rather than to a copy of it.

In some cases, you might want to explicitly pass a variable by reference rather than just passing its value. In that case, you need to modify the function signature as this is a change to its very definition rather than something that can be modified when the function is called. Example 3-4 illustrates how the `increment()` function would change to pass `$number` by reference instead of by value.

Example 3-4. Passing variables by reference

```
function increment(&$number)
{
    $number += 1;

    return $number;
}
```

```
$number = 6;

echo increment($number); // 7
echo $number; // 7
```

In reality, the variable name doesn't need to match both inside and outside the function. I'm using `$number` in both cases here to illustrate the difference in scoping. If you stored an integer in `$a` and passed that variable instead as `increment($a)`, the result would be identical to that in Example 3-4.

See Also

PHP reference documentation on user-defined functions and passing variables by reference.

3.2 Setting a Function's Default Parameters

Problem

You want to set a default value for a function's parameter so invocations don't have to pass it.

Solution

Assign a default value within the function signature itself. For example:

```
function get_book_title($isbn, $error = 'Unable to quer
{
    try {
        $connection = get_database_connection();
        $book = query_isbn($connection, $isbn);

        return $book->title;
    } catch {
        return $error;
    }
}
```

```
get_book_title('978-1-098-12132-7');
```

Discussion

The example in the Solution attempts to query a database for the title of a book based on its ISBN. If the query fails for any reason, the function will return the string passed into the `$error` parameter instead.

To make this parameter optional, the function signature assigns a default value. When calling `get_book_title()` with a single parameter, the default `$error` value is used automatically. You alternatively have the option to pass your own string into this variable when invoking the function, such as `get_book_title(978-1-098-12132-7, Oops!);` .

When defining a function with default parameters, it's a best practice to place all parameters with default values *last* in the function signature. While defining parameters in any order is *possible*, doing so makes it difficult to call the function properly.

Example 3-5 illustrates the kinds of problems that can come up by placing optional parameters before required ones.

WARNING

It is possible to define function parameters with specific defaults in any order. However, declaring mandatory parameters after optional ones is deprecated as of PHP 8.0. Continuing to do so might result in an error in a future version of PHP.

Example 3-5. Misordered default parameters

```
function brew_latte($flavor = 'unflavored', $shots)
{
    return "Brewing a {$shots}-shot, {$flavor} latte!";
}

brew_latte('vanilla', 2);
                         ❶

brew_latte(3);
```

Proper execution. Returns `Brewing a 2-shot, vanilla latte!`

Triggers an `ArgumentCountError` exception because `$shots` is undefined.

In some cases, placing the parameters themselves in a particular order might make logical sense (to make the code more readable, for example). Know that if any parameters are required, every parameter to their left is also effectively required even if you try to define a default value.

See Also

Examples of default arguments in the PHP Manual.

3.3 Using Named Function Parameters

Problem

You want to pass arguments into a function based on the name of the parameter rather than its position.

Solution

Use the named argument syntax while calling a function as follows:

```
array_fill(start_index: 0, count: 100, value: 50);
```

Discussion

By default, PHP leverages positional parameters in function definitions. The Solution example references the native `array_fill()` function that has the following function signature:

```
array_fill(int $start_index, int $count, mixed $value):
```

Basic PHP coding must supply arguments to `array_fill()` in the same order in which they're defined— `$start_index` followed by `$count` followed by `$value`. While the order itself is not a problem, making sense of the meaning of each value when scanning visually through code can be a challenge. Using the basic, ordered parameters, the Solution example would be written as follows, requiring deep familiarity with the function signature to know which integer represents which parameter:

```php
array_fill(0, 100, 50);
```

Named function parameters disambiguate which value is being assigned to which internal variable. They also allow for arbitrary reordering of parameters when you invoke the function as that invocation is now explicit as to which value is assigned to which parameter.

Another key advantage of named arguments is that optional arguments can be *skipped* entirely during function invocation. Consider a verbose activity logging function like in Example 3-6, where multiple parameters are considered optional as they set defaults.

Example 3-6. Verbose activity logging function

```php
activity_log(
    string    $update_reason,
    string    $note           = '',
    string    $sql_statement  = '',
    string    $user_name      = 'anonymous',
    string    $ip_address     = '127.0.0.1',
    ?DateTime $time           = null
): void
```

Internally, Example 3-6 will use its default values when it's called with a single argument; if `$time` is `null`, the value will be silently replaced with a new `DateTime` instance representing "now." However, sometimes you might want to populate one of these optional parameters without wanting to explicitly set *all* of them.

Say you want to replay previously witnessed events from a static log file. User activity was anonymous (so the defaults for `$user_name` and `$ip_address` are adequate), but you need to explicitly set the date at which an event occurred. Without named arguments, an invocation in this case would look similar to Example 3-7.

Example 3-7. Invoking the verbose `activity_log()` function

```
activity_log(
    'Testing a new system',
    '',
    '',
    'anonymous',
    '127.0.0.1',
    new DateTime('2021-12-20')
);
```

With named arguments, you can skip setting parameters to their defaults and explicitly set just the parameters you need to. The preceding code can be simplified to the following:

```
activity_log('Testing a new system', time: new DateTime
```

In addition to drastically simplifying the usage of `activity_log()`, named parameters have the added benefit of keeping your code DRY. The default values for your arguments are stored directly in the function definition rather than being copied to every invocation of the function as well. If you later need to change a default, you can edit the function definition alone.

See Also

The original RFC proposing named parameters.

3.4 Enforcing Function Argument and Return Typing

Problem

You want to force your program to implement type safety and avoid PHP's native loose type comparisons.

Solution

Add input and return types to function definitions. Optionally, add a strict type declaration to the top of each file to enforce values matching type annotations (and emit a fatal error if they don't match). For example:

```php
declare(strict_types=1);

function add_numbers(int $left, int $right): int
{
    return $left + $right;
}

add_numbers(2, 3); ❶

add_numbers(2, '3'); ❷
```

❶ This is a perfectly valid operation and will return the integer `5`.

❷ While `2 + '3'` is valid PHP code, the string `'3'` violates the function's type definitions and will trigger a fatal error.

Discussion

PHP natively supports various scalar types and allows developers to declare both function input parameters and returns to identify the kinds of values that are allowable for each. In addition, developers can specify their own custom classes and interfaces as types, or leverage class inheritance within the type system.[1]

Parameter types are annotated by placing the type directly before the name of the parameter when defining the function. Similarly, return types are specified by appending the function signature with a `:` and the type that function would return as in the following:

```php
function name(type $parameter): return_type
{
    // ...
}
```

Table 3-1 enumerates the simplest types leveraged by PHP.

Table 3-1. Simple single types in PHP

Type	Description
`array`	The value must be an array (containing any type of values).
`callable`	The value must be a callable function.
`bool`	The value must be a Boolean.
`float`	The value must be a floating-point number.
`int`	The value must be an integer.
`string`	The value must be a string.
`iterable`	The value must be an array or an object that implements `Traversable` .
`mixed`	The object can be any value.
`void`	A return-only type indicating that the function does not return a value.
`never`	A return-only type indicating a function does not return; it either calls `exit` , throws an exception, or is intentionally an infinite loop.

In addition, both built-in and custom classes can be used to define types, as shown in Table 3-2.

Table 3-2. Object types in PHP

Type	Description
Class/interface name	The value must be an instance of the specified class or implementation of an interface.

Type	Description
`self`	The value must be an instance of the same class as the one in which the declaration is used.
`parent`	The value must be an instance of the parent of the class in which the declaration is used.
`object`	The value must be an instance of an object.

PHP also permits simple scalar types to be expanded by either making them nullable or combining them into *union types*. To make a specific type nullable, you have to prefix the type annotation with a `?`. This will instruct the compiler to allow values to be either the specified type or `null`, as in Example 3-8.

Example 3-8. Function utilizing nullable parameters

```
function say_hello(?string $message): void
{
    echo 'Hello, ';

    if ($message === null) {
        echo 'world!';
    } else {
        echo $message . '!';
    }
}

say_hello('Reader'); // Hello, Reader!
say_hello(null); // Hello, world!
```

A union type declaration combines multiple types into a single declaration by concatenating simple types together with the pipe character (`|`). If you were to rewrite the type declarations on the Solution example with a union type combining strings and integers, the fatal error thrown by passing in a string for addition would resolve itself. Consider the possible rewrite in Example 3-9 that would permit *either* integers or strings as parameters.

Example 3-9. Rewriting the Solution example to leverage union types

```
function add_numbers(int|string $left, int|string $rig
{
    return $left + $right;
}

add_numbers(2, '3'); // 5
```

The biggest problem with this alternative is that adding strings together with the `+` operator has no meaning in PHP. If both parameters are numeric (either integers or integers represented as strings), the function will work just fine. If either is a non-numeric string, PHP will throw a `TypeError` as it doesn't know how to "add" two strings together. These kinds of errors are what you hope to avoid by adding type declarations to your code and enforcing strict typing—they formalize the contract you expect your code to support and encourage programming practices that naturally defend against coding mistakes.

By default, PHP uses its typing system to *hint* at which types are allowed into and returned from functions. This is useful to prevent passing bad data into a function, but it relies heavily on either developer diligence or additional tooling[2] to enforce typing. Rather than rely on humans' ability to check code, PHP allows for a static declaration in each file that all invocations should follow strict typing.

Placing `declare(strict_types=1);` at the top of a file tells the PHP compiler you intend for all invocations in that file to obey parameter and return type declarations. Note that this directive applies to *invocations* within the file where it's used, not to the definitions of functions in that file. If you call functions from another file, PHP will honor the type declarations in that file as well. However, placing this directive in your file will not force other files that reference your functions to obey the typing system.

See Also

PHP documentation on type declarations and the `declare` construct.

3.5 Defining a Function with a Variable Number of Arguments

Problem

You want to define a function that takes one or more arguments without knowing ahead of time how many values will be passed in.

Solution

Use PHP's spread operator (...) to define a variable number or arguments:

```
function greatest(int ...$numbers): int
{
    $greatest = 0;
    foreach ($numbers as $number) {
        if ($number > $greatest) {
            $greatest = $number;
        }
    }

    return $greatest;
}

greatest(7, 5, 12, 2, 99, 1, 415, 3, 7, 4);
// 415
```

Discussion

The *spread operator* automatically adds all parameters passed in that particular position or after it to an array. This array can be typed by prefixing the spread operator with a type declaration (review Recipe 3.4 for more on typing), thus requiring every element of the array to match a specific type. Invoking the function defined in the Solution example as `greatest(2, "five");` will throw a `TypeError`, as you have explicitly declared an `int` type for every member of the `$numbers` array.

Your function can accept more than one positional parameter while still leveraging the spread operator to accept an unlimited number of additional arguments. The function defined in Example 3-10 will print a greeting to the screen for an unlimited number of individuals.

Example 3-10. Utilizing the spread operator

```
function greet(string $greeting, string ...$names): voi
{
    foreach($names as $name) {
        echo $greeting . ', ' . $name . PHP_EOL;
    }
}

greet('Hello', 'Tony', 'Steve', 'Wanda', 'Peter');
// Hello, Tony
// Hello, Steve
// Hello, Wanda
// Hello, Peter

greet('Welcome', 'Alice', 'Bob');
// Welcome, Alice
// Welcome, Bob
```

The spread operator has more utility than just function definition. While it can be used to pack multiple arguments into an array, it can also be used to unpack an array into multiple arguments for a more traditional function invocation. Example 3-11 provides a trivial illustration of how this array unpacking works by using the spread operator to pass an array into a function that does not accept an array.

Example 3-11. Unpacking an array with the spread operator

```
function greet(string $greeting, string $name): void
{
    echo $greeting . ', ' . $name  . PHP_EOL;
}

$params = ['Hello', 'world'];
greet(...$params);
// Hello, world
```

In some cases, a more complex function might return multiple values (as discussed in the next recipe), so passing the return of one function into another becomes simple with the spread operator. In fact, any array or variable that implements PHP's Traversable interface can be unpacked into a function invocation in this manner.

See Also

PHP documentation on variable-length argument lists.

3.6 Returning More Than One Value

Problem

You want to return multiple values from a single function invocation.

Solution

Rather than returning a single value, return an array of multiple values and unpack them by using `list()` outside the function:

```php
function describe(float ...$values): array
{
    $min = min($values);
    $max = max($values);
    $mean = array_sum($values) / count($values);

    $variance = 0.0;
    foreach($values as $val) {
        $variance += pow(($val - $mean), 2);
    }
    $std_dev = (float) sqrt($variance/count($values));

    return [$min, $max, $mean, $std_dev];
}

$values = [1.0, 9.2, 7.3, 12.0];
list($min, $max, $mean, $std) = describe(...$values);
```

Discussion

PHP is only capable of returning one value from a function invocation, but that value itself could be an array containing multiple values. When paired with PHP's `list()` construct, this array can be easily destructured to individual variables for further use by the program.

While the need to return many different values isn't common, when the occasion comes up, being able to do so can be incredibly handy. One example is in web authentication. Many modern systems today use JSON Web Tokens (JWTs), which are period-delimited strings of Base64-encoded data. Each component of a JWT represents a separate, discrete thing—a header describing the algorithm used, the data in the token payload, and a verifiable signature on that data.

When reading a JWT as a string, PHP applications often leverage the built-in `explode()` function to split the string on the periods delimiting each component. A simple use of `explode()` might appear as follows:

```
$jwt_parts = explode('.', $jwt);
$header = base64_decode($jwt_parts[0]);
$payload = base64_decode($jwt_parts[1]);
$signature = base64_decode($jwt_parts[2]);
```

The preceding code works just fine, but the repeated references to positions within an array can be difficult to follow both during development and debugging later if a problem arises. In addition, developers must manually decode every part of the JWT separately; forgetting to invoke `base64_decode()` could be fatal to the operation of the program.

An alternative approach is to unpack and automatically decode the JWT within a function and return an array of the components, as shown in Example 3-12.

Example 3-12. Decoding a JWT

```
function decode_jwt(string $jwt): array
{
    $parts = explode('.', $jwt);
```

```
        return array_map('base64_decode', $parts);
    }

    list($header, $payload, $signature) = decode_jwt($jwt);
```

A further advantage of using a function to unpack a JWT rather than decomposing each element directly is that you could build in automated signature verification or even filter JWTs for acceptability based on the encryption algorithms declared in the header. While this logic could be applied procedurally while processing a JWT, keeping everything in a single function definition leads to cleaner, more maintainable code.

The biggest drawback to returning multiple values in one function call is in typing. These functions have an `array` return type, but PHP doesn't natively allow for specifying the type of the elements within an array. We have potential workarounds to this limitation by way of documenting the function signature and integrating with a static analysis tool like Psalm or PHPStan, but we have no native support within the language for typed arrays. As such, if you're using strict typing (and you *should* be), returning multiple values from a single function invocation should be a rare occurrence.

See Also

Recipe 3.5 on passing a variable number of arguments and Recipe 1.3 for more on PHP's `list()` construct. Also reference the phpDocumentor documentation on typed arrays that can be enforced by tools like Psalm.

3.7 Accessing Global Variables from Within a Function

Problem

Your function needs to reference a globally defined variable from elsewhere in the application.

Solution

Prefix any global variables with the `global` keyword to access them within the function's scope:

```
$counter = 0;

function increment_counter()
{
    global $counter;

    $counter += 1;
}

increment_counter();

echo $counter; // 1
```

Discussion

PHP separates operations into various scopes based on the context in which a variable is defined. For most programs, a single scope spans all included or required files. A variable defined in this global scope is available *everywhere* regardless of which file is currently executing, as demonstrated in Example 3-13.

Example 3-13. Variables defined in the global scope are available to included scripts

```
$apple = 'honeycrisp';

include 'someotherscript.php';
                              ❶
```

❶ The `$apple` variable is also defined within this script and available for use.

User-defined functions, however, define their own scope. A variable defined outside a user-defined function is *not available* within the body of the function. Likewise, any variable defined within the function is not available outside the function. Example 3-14 illustrates the boundaries of the parent and function scope in a program.

Example 3-14. Local versus global scoping

```
$a = 1;                                  ❶

function example(): void
{
    echo $a . PHP_EOL;                   ❷

    $a = 2;                              ❸

    $b = 3;                              ❹
}

example();

echo $a . PHP_EOL;                       ❺

echo $b . PHP_EOL;                       ❻
```

❶ The variable `$a` is initially defined in the parent scope.

❷ Inside the function scope, `$a` is not yet defined. Attempting to `echo` its value will result in a warning.

❸ Defining a variable called `$a` within the function will *not* overwrite the value of the same-named variable outside the function.

❹ Defining a variable called `$b` within the function makes it available within the function, but this value will not escape the scope of the function.

❺ Echoing `$a` outside the function, even after invoking `example()`, will print the initial value you've set, as the function did not change the variable's value.

❻ Since `$b` was defined within the function, it is undefined in the scope of the parent application.

NOTE

It is possible to pass a variable into a function call *by reference* if the function is defined to accept a variable in such a way. However, this is a decision made by the definition of the function and not a runtime flag available to routines leveraging that function after the fact. Example 3-4 shows what pass-by-reference might look like.

To reference variables defined outside its scope, a function needs to declare those variables as *global* within its own scope. To reference the parent scope, you can rewrite Example 3-14 as Example 3-15.

Example 3-15. Local versus global scoping, revisited

```
$a = 1;

function example(): void
{
    global $a, $b;            ❶

    echo $a . PHP_EOL;        ❷

    $a = 2;                   ❸

    $b = 3;                   ❹
}

example();

echo $a . PHP_EOL;            ❺

echo $b . PHP_EOL;            ❻
```

❶ By declaring both `$a` and `$b` to be global variables, you are telling the function to use values from the parent scope rather than its own scope.

❷ With a reference to the *global* `$a` variable, you can now actually print it to output.

Likewise, any changes to `$a` within the scope of the function will

3. Likewise, any changes to $a within the scope of the function will impact the variable in the parent scope.

4. Similarly, you now define `$b` but, as it's global, this definition will bubble out to the parent scope as well.

5. Echoing `$a` will now reflect the changes made within the scope of `example()` as you made the variable global.

6. Likewise, `$b` is now defined globally and can be echoed to output as well.

There is no limit on the number of global variables PHP can support aside from the memory available to the system. Additionally, *all* globals can be listed by enumerating the special `$GLOBALS` array defined by PHP. This associative array contains references to all variables defined within the global scope. This special array can be useful if you want to reference a specific variable in the global scope *without* declaring the variable as global, as in Example 3-16.

Example 3-16. Using the associative `$GLOBALS` array

```php
$var = 'global';

function example(): void
{
    $var = 'local';

    echo 'Local variable: ' . $var . PHP_EOL;
    echo 'Global variable: ' . $GLOBALS['var'] . PHP_E
}

example();
// Local variable: local
// Global variable: global
```

WARNING

As of PHP 8.1, it is no longer possible to overwrite the entirety of the `$GLOBALS` array. In previous versions, you could reset it to an empty array (for example, during test runs of your code). Moving forward, you can edit only the contents of the array rather than manipulating the collection in its entirety.

Global variables are a handy way to reference state across your application, but they can lead to confusion and maintainability issues if overused. Some large applications leverage global variables heavily—WordPress, a PHP-based project that powers more than 40% of the internet,[3] uses global variables throughout its codebase. However, most application developers agree that global variables should be used sparingly, if at all, to help keep systems clean and easy to maintain.

See Also

PHP documentation on variable scope and the special `$GLOBALS` array.

3.8 Managing State Within a Function Across Multiple Invocations

Problem

Your function needs to keep track of its change in state over time.

Solution

Use the `static` keyword to define a locally scoped variable that retains its state between function invocations:

```
function increment()
{
    static $count = 0;

    return $count++;
}

echo increment(); // 0
echo increment(); // 1
echo increment(); // 2
```

Discussion

A static variable exists only within the scope of the function in which it is declared. However, unlike regular local variables, it holds on to its value every time you return to the scope of the function. In this way, a function can become *stateful* and keep track of certain data (like the number of times it's been called) between independent invocations.

In a typical function, using the `=` operator will assign a value to a variable. When the `static` keyword is applied, this assignment operation only happens the first time that function is called. Subsequent calls will reference the previous state of the variable and allow the program to either use or modify the stored value as well.

One of the most common use cases of static variables is to track the state of a recursive function. Example 3-17 demonstrates a function that recursively calls itself a fixed number of times before exiting.

Example 3-17. Using a static variable to limit recursion depth

```
function example(): void
{
    static $count = 0;

    if ($count >= 3) {
        $count = 0;
        return;
    }

    $count += 1;

    echo 'Running for loop number ' . $count . PHP_EOL;
    example();
}
```

The `static` keyword can also be used to keep track of expensive resources that might be needed by a function multiple times but that you might only want a single instance of. Consider a function that logs messages to a database: you might not be able to pass a database connection into the function itself, but you want to ensure that the function only opens a *single* database connection. Such a logging function might be implemented as in Example 3-18.

Example 3-18. Using a static variable to hold a database connection

```
function logger(string $message): void
{
    static $dbh = null;
    if ($dbh === null) {
        $dbh = new PDO(DATABASE_DSN, DATABASE_USER, DAT
    }

    $sql = 'INSERT INTO messages (message) VALUES (:mes
    $statement = $dbh->prepare($sql);

    $statement->execute([':message', $message]);
}

logger('This is a test');
                          ❶

logger('This is another message');
                          ❷
```

❶ The first time `logger()` is called, it will define the value of the static `$dbh` variable. In this case, it will connect to a database by using the PHP Data Objects (PDO) interface. This interface is a standard object provided by PHP for accessing databases.

❷ Every subsequent call to `logger()` will leverage the initial connection opened to the database and stored in `$dbh`.

Note that PHP automatically manages its memory usage and automatically clears variables from memory when they leave scope. For regular variables within a function, this means the variables are freed from memory as soon as the function completes. Static and global variables are *never* cleaned up until the program itself exits, as they are always in scope. Take care when using the `static` keyword to ensure that you aren't storing unnecessarily large pieces of data in memory. In Example 3-18, you open a connection to a database that will never be automatically closed by the function you've created.

While the `static` keyword can be a powerful way to reuse state across function calls, it should be used with care to ensure that your application doesn’t do anything unexpected. In many cases, it might be better to explicitly pass variables representing state into the function. Even better would be to encapsulate the function’s state as part of an overarching object, which is covered in Chapter 8.

See Also

PHP documentation on variable scoping, including the `static` keyword.

3.9 Defining Dynamic Functions

Problem

You want to define an anonymous function and reference it as a variable within your application because you only want to use or call the function a single time.

Solution

Define a closure that can be assigned to a variable and passed into another function as needed:

```php
$greet = function($name) {
    echo 'Hello, ' . $name . PHP_EOL;
};

$greet('World!');
// Hello, World!
```

Discussion

Whereas most functions in PHP have defined names, the language supports the creation of unnamed (so-called *anonymous*) functions, also called *closures* or *lambdas*. These functions can encapsulate either simple or complex logic and can be assigned directly to variables for reference elsewhere in the program.

Internally, anonymous functions are implemented using PHP's native `Closure` class. This class is declared as `final`, which means no class can extend it directly. Yet, anonymous functions are all instances of this class and can be used either directly as functions or as objects.

By default, closures do not inherit any scope from the parent application and, like regular functions, define variables within their own scope. Variables from the parent scope can be passed directly into a closure by leveraging the `use` directive when defining a function. Example 3-19 illustrates how variables from one scope can be passed into another dynamically.

Example 3-19. Passing a variable between scopes with `use()`

```php
$some_value = 42;

$foo = function() {
    echo $some_value;
};

$bar = function() use ($some_value) {
    echo $some_value;
};

$foo(); // Warning: Undefined variable

$bar(); // 42
```

Anonymous functions are used in many projects to encapsulate a piece of logic for application against a collection of data. The next recipe covers exactly that use case.

NOTE

Older versions of PHP used `create_function()` for similar utility. Developers could create an anonymous function as a string and pass that code into `create_function()` to turn it into a closure instance. Unfortunately, this method used `eval()` under the hood to evaluate the string—a practice considered highly unsafe. While some older projects might still use `create_function()`, the function itself was deprecated in PHP 7.2 and removed from the language entirely in version 8.0.

See Also

PHP documentation on anonymous functions.

3.10 Passing Functions as Parameters to Other Functions

Problem

You want to define part of a function's implementation and pass that implementation as an argument to another function.

Solution

Define a closure that implements part of the logic you need and pass that directly into another function as if it were any other variable:

```
$reducer = function(?int $carry, int $item): int {
    return $carry + $item;
};

function reduce(array $array, callable $callback, ?int
{
    $acc = $initial;
    foreach ($array as $item) {
        $acc = $callback($acc, $item);
    }

    return $acc;
}

$list = [1, 2, 3, 4, 5];
$sum = reduce($list, $reducer); // 15
```

Discussion

PHP is considered by many to be a *functional language*, as functions are first-class elements in the language and can be bound to variable names, passed as arguments, or even returned from other functions. PHP supports functions as variables through the callable type as implemented in the language. Many core functions (like `usort()`, `array_map()`, and `array_reduce()`) support passing a callable parameter, which is then used internally to define the function's overall implementation.

The `reduce()` function defined in the Solution example is a user-written implementation of PHP's native `array_reduce()` function. Both have the same behavior, and the Solution could be rewritten to pass `$reducer` directly into PHP's native implementation with no change in the result:

```
$sum = array_reduce($list, $reducer); // 15
```

Since functions can be passed around like any other variable, PHP has the ability to define partial implementations of functions. This is achieved by defining a function that, in turn, returns another function that can be used elsewhere in the program.

For example, you can define a function to set up a basic multiplier routine that multiplies any input by a *fixed* base amount, as in Example 3-20. The main function returns a new function each time you call it, so you can create functions to double or triple arbitrary values and use them however you want.

Example 3-20. Partially applied multiplier function

```
function multiplier(int $base): callable
{
    return function(int $subject) use ($base): int {
        return $base * $subject;
    };
}

$double = multiplier(2);
$triple = multiplier(3);

$double(6);  // 12
$double(10); // 20
$triple(3);  // 9
$triple(12); // 36
```

Breaking functions apart like this is known as *https://oreil.ly/-a4l[_currying*]. This is the practice of changing a function with multiple input parameters into a series of functions, that each take a *single* parameter, with most of those parameters being functions themselves. To fully illustrate how this can work in PHP, let's look at Example 3-21 and walk through a rewrite of the `multiplier()` function.

Example 3-21. Walk-through of currying in PHP

```
function multiply(int $x, int $y): int ❶
{
    return $x * $y;
}

multiply(7, 3); // 21

function curried_multiply(int $x): callable ❷
{
    return function(int $y) use ($x): int { ❸
        return $x * $y; ❹
    };
}

curried_multiply(7)(3); // 21 ❺
```

❶ The most basic form of the function takes two values, multiplies them together, and returns a final result.

❷ When you curry the function, you want each component function to only take a single value. The new `curried_multiply()` only accepts one parameter but returns a function that leverages that parameter internally.

❸ The internal function references the value passed by your previous function invocation automatically (with the `use` directive).

❹ The resulting function implements the same business logic as the basic form.

Calling a curried function has the appearance of calling *multiple* functions in series, but the result is the same.

The biggest advantage of currying, as in Example 3-21, is that a partially applied function can be passed around as a variable and used elsewhere. Similar to using the `multiplier()` function, you can create a doubling or tripling function by *partially* applying your curried multiplier as follows:

```
$double = curried_multiply(2);
$triple = curried_multiply(3);
```

Partially applied, curried functions are themselves callable functions but can be passed into other functions as variables and fully invoked later.

See Also

Details on anonymous functions in Recipe 3.9.

3.11 Using Concise Function Definitions (Arrow Functions)

Problem

You want to create a simple, anonymous function that references the parent scope without verbose `use` declarations.

Solution

Use PHP's short anonymous function (arrow function) syntax to define a function that inherits its parent's scope automatically:

```
$outer = 42;

$anon = fn($add) => $outer + $add;

$anon(5); // 47
```

Discussion

Arrow functions were introduced in PHP 7.4 as a way to write more concise anonymous functions, as in Recipe 3.9. Arrow functions automatically capture any referenced variables and import them (by value rather than by reference) into the scope of the function.

A more verbose version of the Solution example could be written as shown in Example 3-22 while still achieving the same level of functionality.

Example 3-22. Long form of an anonymous function

```
$outer = 42;

$anon = function($add) use ($outer) {
    return $outer + $add;
};

$anon(5);
```

Arrow functions always return a value—it is impossible to either implicitly or explicitly return `void`. These functions follow a very specific syntax and always return the result of their expression: `fn (arguments) => expression`. This structure makes arrow functions useful in a wide variety of situations.

One example is a concise inline definition of a function to be applied to all elements in an array via PHP's native `array_map()`. Assume input user data is an array of strings that each represent an integer value and you want to convert the array of strings into an array of integers to enforce proper type safety. This can easily be accomplished via Example 3-23.

Example 3-23. Convert an array of numeric strings to an array of integers

```
$input = ['5', '22', '1093', '2022'];

$output = array_map(fn($x) => intval($x), $input);
// $output = [5, 22, 1093, 2022]
```

Arrow functions only permit a single-line expression. If your logic is complicated enough to require multiple expressions, use a standard anonymous function (see Recipe 3.9) or define a named function in your code. This being said, an arrow function itself is an expression, so one arrow function can actually return another.

The ability to return an arrow function as the expression of another arrow function leads to a way to use arrow functions in *curried* or partially applied functions to encourage code reuse. Assume you want to pass a function in the program that performs modulo arithmetic with a fixed modulus. You can do so by defining one arrow function to perform the calculation and wrap it in another that specifies the modulus, assigning the final, curried function to a variable you can use elsewhere, as in Example 3-24.

NOTE

Modulo arithmetic is used to create *clock functions* that always return a specific set of integer values regardless of the integer input. You take the modulus of two integers by dividing them and returning the integer remainder. For example, "12 modulo 3" is written as `12 % 3` and returns the remainder of `12/3`, or `0`. Similarly, "15 modulo 6" is written as `15 % 6` and returns the remainder of `15/6`, or `3`. The return of a modulo operation is never greater than the modulus itself (`3` or `6` in the previous two examples, respectively). Modulo arithmetic is commonly used to group large collections of input values together or to power cryptographic operations, which are discussed further in Chapter 9.

Example 3-24. Function currying with arrow functions

```
$modulo = fn($x) => fn($y) => $y % $x;

$mod_2 = $modulo(2);
$mod_5 = $modulo(5);

$mod_2(15); // 1
$mod_2(20); // 0
$mod_5(12); // 2
$mod_5(15); // 0
```

Finally, just like regular functions, arrow functions can accept multiple arguments. Rather than passing a single variable (or implicitly referencing variables defined in the parent scope), you can just as easily define a function with multiple parameters and freely use them within the expression. A trivial equality function might use an arrow function as follows:

```
$eq = fn($x, $y) => $x == $y;

$eq(42, '42'); // true
```

See Also

Details on anonymous functions in Recipe 3.9 and the PHP Manual documentation on arrow functions.

3.12 Creating a Function with No Return Value

Problem

You need to define a function that does not return data to the rest of the program after it completes.

Solution

Use explicit type declarations and reference the `void` return type:

```
const MAIL_SENDER = 'wizard@oz.example';
const MAIL_SUBJECT = 'Incoming from the Wonderful Wizar

function send_email(string $to, string $message): void
{
    $headers = ['From' => MAIL_SENDER];

    $success = mail($to, MAIL_SUBJECT, $message, $heade

    if (!$success) {
        throw new Exception('The man behind the curtair
    }
}
```

```
send_email('dorothy@kansas.example', 'Welcome to the Em
```

Discussion

The Solution example uses PHP's native `mail()` function to dispatch a simple message with a static subject to the specified recipient. PHP's `mail()` returns either `true` (on success) or `false` (when there's an error). In the Solution example, you merely want to throw an exception when something goes wrong but otherwise want to return silently.

NOTE

In many cases, you might want to return a flag—a Boolean value or a string or `null` —when a function completes to indicate what has happened so the rest of your program can behave appropriately. Functions that return *nothing* are relatively rare, but they do come up when your program is communicating with an outside party and the result of that communication doesn't impact the rest of the program. Sending a fire-and-forget connection to a message queue or logging to the system error log are both common use cases for a function that returns `void` .

The `void` return type is enforced on compile time in PHP, meaning your code will trigger a fatal error if the function body returns *anything* at all, even if you haven't executed anything yet. Example 3-25 illustrates both valid and invalid uses of `void` .

Example 3-25. Valid and invalid uses of the `void` return type

```
function returns_scalar(): void
{
    return 1;
                      ❶

}

function no_return(): void
{

                      ❷

}
```

```
function empty_return(): void
{
    return;
                               ❸

}

function returns_null(): void
{
    return null;
                               ❹

}
```

❶ Returning a scalar type (such as a string, integer, or Boolean) will trigger a fatal error.

❷ Omitting any kind of return in a function is valid.

❸ Explicitly returning no data is valid.

❹ Even though `null` is "empty," it still counts as a return and will trigger a fatal error.

Unlike most other types in PHP, the `void` type is only valid for returns. It cannot be used as a parameter type in a function definition; attempts to do so will result in a fatal error at compile time.

See Also

The original RFC introducing the `void` return type in PHP 7.1.

3.13 Creating a Function That Does Not Return

Problem

You need to define a function that explicitly exits and to ensure that other parts of your application are aware it will never return.

Solution

Use explicit type annotations and reference the `never` return type. For example:

```
function redirect(string $url): never
{
    header("Location: $url");
    exit();
}
```

Discussion

Some operations in PHP are intended to be the last action the engine takes before exiting the current process. Calling `header()` to define a specific response header must be done prior to printing any body to the response itself. Specifically, calling `header()` to trigger a redirect is usually the last thing you want your application to do—printing any body text or processing any other operation after you've told the requesting client to redirect elsewhere has no meaning or value.

The `never` return type signals both to PHP and to other parts of your code that the function is *guaranteed* to halt the program's execution by way of either `exit()` or `die()` or throwing an exception.

If a function that leverages the `never` return type still returns implicitly, as in Example 3-26, PHP will throw a `TypeError` exception.

Example 3-26. Implicit return in a function that should never return

```
function log_to_screen(string $message): never
{
    echo $message;
}
```

Likewise, if a `never`-typed function *explicitly* returns a value, PHP will throw a `TypeError` exception. In both situations, whether an implicit or an explicit return, this exception is enforced at call time (when the function is invoked) rather than when the function is defined.

See Also

The original RFC introducing the `never` return type in PHP 8.1.

[1] Custom classes and objects are discussed at length in Chapter 8.

[2] PHP CodeSniffer is a popular developer tool for automatically scanning a codebase and ensuring that all code matches a specific coding standard. It can be trivially extended to enforce a strict type declaration in all files as well.

[3] The market reach of WordPress was about 63% of websites using content management systems and more than 43% of all websites as of March 2023 according to W3Techs.