Chapter 4. Strings

Strings are one of the fundamental building blocks of data in PHP. Each string represents an ordered sequence of bytes. Strings can range from human-readable sections of text (like `To be or not to be`) to sequences of raw bytes encoded as integers (such as `\110\145\154\154\157\40\127\157\162\154\144\41`).[1] Every element of data read or written by a PHP application is represented as a string.

In PHP, strings are typically encoded as ASCII values, although you can convert between ASCII and other formats (like UTF-8) as necessary. Strings can contain `null` bytes when needed and are essentially limitless in terms of storage so long as the PHP process has adequate memory available.

The most basic way to create a string in PHP is with single quotes. Single-quoted strings are treated as literal statements—there are no special characters or any kind of *interpolation* of variables. To include a literal single quote within a single-quoted string, you must *escape* that quote by prefixing it with a backslash—for example, `\'`. In fact, the only two characters that need to be—or even can be—escaped are the single quote itself or the backslash. Example 4-1 shows single-quoted strings along with their corresponding printed output.

NOTE

Variable interpolation is the practice of referencing a variable directly by name within a string and letting the interpreter replace the variable with its value at runtime. Interpolation allows for more flexible strings, as you can write a single string but dynamically replace some of its contents to fit the context of its location in code.

Example 4-1. Single-quoted strings

```
print 'Hello, world!';
// Hello, world!
```

```
print 'You\'ve only got to escape single quotes and the
// You've only got to escape single quotes and the \ ch

print 'Variables like $blue are printed as literals.';
// Variables like $blue are printed as literals.

print '\110\145\154\154\157\40\127\157\162\154\144\41';
// \110\145\154\154\157\40\127\157\162\154\144\41
```

More complicated strings might need to interpolate variables or reference special characters, like a newline or tab. For these more complicated use cases, PHP requires the use of double quotes instead and allows for various escape sequences, as shown in Table 4-1.

Table 4-1. Double-quoted string escape sequences

Escape sequence	Character	Example
`\n`	Newline	`"This string ends in a new line.\n"`
`\r`	Carriage return	`"This string ends with a carriage return.\r"`
`\t`	Tab	`"Lots\tof\tspace"`
`\\`	Backslash	`"You must escape the \\ character."`
`\$`	Dollar sign	`A movie ticket is \$10.`
`\"`	Double quote	`"Some quotes are \"scare quotes.\""`
`\0` through `\777`	Octal character value	`"\120\110\120"`

Escape sequence	Character	Example
`\x0` through `\xFF`	Hex character value	`"\x50\x48\x50"`

Except for special characters that are explicitly escaped with leading backspaces, PHP will automatically substitute the value of any variable passed within a double-quoted string. Further, PHP will interpolate entire expressions within a double-quoted string if they're wrapped in curly braces (`{}`) and treat them as a variable. Example 4-2 shows how variables, complex or otherwise, are treated within double-quoted strings.

Example 4-2. Variable interpolation within double-quoted strings

```
print "The value of \$var is $var";
❶

print "Properties of objects can be interpolated, too.
❷

print "Prints the value of the variable returned by get
❸
```

❶ The first reference to `$var` is escaped, but the second will be replaced by its actual value. If `$var = 'apple'` , the string will print `The value of $var is apple` .

❷ Using curly braces enables the direct reference of object properties within a double-quoted string as if these properties were locally defined variables.

❸ Assuming `getVar()` returns the name of a defined variable, this line will both execute the function and print the value assigned to that variable.

Both single- and double-quoted strings are represented as single lines. Often, though, a program will need to represent multiple lines of text (or multiple lines of encoded binary) as a string. In such a situation, the best tool at a developer's disposal is a Heredoc.

A *Heredoc* is a literal block of text that's started with three angle brackets (the `<<<` operator), followed by a named identifier, followed by a newline. Every subsequent line of text (including newlines) is part of the string, up to a completely independent line containing nothing but the Heredoc's named identifier and a semicolon. Example 4-3 illustrates how a Heredoc might look in code.

NOTE

The identifier used for a Heredoc does not need to be capitalized. However, it is common convention in PHP to always capitalize these identifiers to help distinguish them from the text definition of the string.

Example 4-3. String definition using the Heredoc syntax

```
$poem = <<<POEM
To be or not to be,
That is the question
POEM;
```

Heredocs function just like double-quoted strings and allow variable interpolation (or special characters like escaped hexadecimals) within them. This can be particularly powerful when encoding blocks of HTML within an application, as variables can be used to make the strings dynamic.

In some situations, though, you might want a string literal rather than something open to variable interpolation. In that case, PHP's Nowdoc syntax provides a single-quoted style alternative to Heredoc's double-quoted string analog. A Nowdoc looks almost exactly like a Heredoc, except the identifier itself is enclosed in single quotes, as in Example 4-4.

Example 4-4. String definition using the Nowdoc syntax

```
$poem = <<<'POEM'
To be or not to be,
That is the question
POEM;
```

Both single and double quotes can be used within Heredoc and Nowdoc blocks without additional escaping. Nowdocs, however, will not interpolate or dynamically replace any values, whether they are escaped or otherwise.

The recipes that follow help further illustrate how strings can be used in PHP and the various problems they can solve.

4.1 Accessing Substrings Within a Larger String

Problem

You want to identify whether a string contains a specific substring. For example, you want to know if a URL contains the text `/secret/`.

Solution

Use `strpos()`:

```
if (strpos($url, '/secret/') !== false) {
    // A secret fragment was detected, run additional l
    // ...
}
```

Discussion

PHP's `strpos()` function will scan through a given string and identify the starting position of the first occurrence of a given substring. This function literally looks for a needle in a haystack, as the function's arguments are named `$haystack` and `$needle`, respectively. If the substring (`$needle`) is not found, the function returns a Boolean `false`.

It's important in this case to use *strict equality comparison*, as `strpos()` will return `0` if the substring appears at the very beginning of the string being searched. Remember from Recipe 2.3 that comparing values with only two equals signs will attempt to recast the types, converting an integer `0` into a

Boolean `false`; always use strict comparison operators (either `===` for equality or `!==` for inequality) to avoid confusion.

If the `$needle` appears multiple times within a string, `strpos()` only returns the position of the first occurrence. You can search for additional occurrences by adding an optional position offset as a third parameter to the function call, as in Example 4-5. Defining an offset also allows you to search the latter part of a string for a substring that you know already appears earlier in the string.

Example 4-5. Count all occurrences of a substring

```
function count_occurrences($haystack, $needle)
{
    $occurrences = 0;
    $offset = 0;
    $pos = 0;
                                ❶


    do {
        $pos = strpos($haystack, $needle, $offset);

        if ($pos !== false) {
                                ❷

            $occurrences += 1;
            $offset = $pos + 1;
                                ❸

        }
    } while ($pos !== false);
                                ❹


    return $occurrences;
}

$str = 'How much wood would a woodchuck chuck if a wooc

print count_occurrences($str, 'wood'); // 4
print count_occurrences($str, 'nutria'); // 0
```

❶ All of the variables are initially set to `0` so you can track new string occurrences.

❷ If and only if the string was found should an occurrence be counted

If and only if the string was found, should an occurrence be counted. ❷

If the string was found, update the offset but also increment by `1` so you don't repeatedly recount the occurrence you've already found. ❸

Once you've reached the last occurrence of the target substring, exit the loop and return the total count. ❹

See Also

PHP documentation on `strpos()`.

4.2 Extracting One String from Within Another

Problem

You want to extract a small string from a much larger string—for example, the domain name from an email address.

Solution

Use `substr()` to select the part of the string you want to extract:

```
$string = 'eric.mann@cookbook.php';
$start = strpos($string, '@');

$domain = substr($string, $start + 1);
```

Discussion

PHP's `substr()` function returns the portion of a given string, based on an initial offset (the second parameter) up to an optional length. The full function signature is as follows:

```
function substr(string $string, int $offset, ?int $leng
```

If the `$length` parameter is omitted, `substr()` will return the entire remainder of the string. If the `$offset` parameter is greater than the length of the input string, an empty string is returned.

You can also specify a *negative* offset to return a subset starting from the end of the string instead of the beginning, as in Example 4-6.

Example 4-6. Substring with a negative offset

```
$substring = substr('phpcookbook', -3);
                                       ❶

$substring = substr('phpcookbook', -2);
                                       ❷

$substring = substr('phpcookbook', -8, 4);
                                       ❸
```

❶ Returns `ook` (the last three characters)

❷ Returns `ok` (the last two characters)

❸ Returns `cook` (the middle four characters)

You should be aware of some other edge cases regarding offsets and string lengths with `substr()`. It is possible for the offset to legitimately start within the string, but for `$length` to run past the end of the string. PHP catches this discrepancy and merely returns the remainder of the original string, even if the final return is *less* than the specified length. Example 4-7 details some potential outputs of `substr()` based on various specified lengths.

Example 4-7. Various substring lengths

```
$substring = substr('Four score and twenty', 11, 3);
                                                   ❶

$substring = substr('Four score and twenty', 99, 3);
                                                   ❷

$substring = substr('Four score and twenty', 20, 3);
                                                   ❸
```

❶ Returns `and`

❷ Returns an empty string

❸ Returns `y`

Returns `y` ❸

Another edge case is a negative `$length` supplied to the function. When requesting a substring with a negative length, PHP will remove that many characters from the substring it returns, as illustrated in Example 4-8.

Example 4-8. Substring with a negative length

```
$substring = substr('Four score and twenty', 5);
                                              ❶

$substring = substr('Four score and twenty', 5, -11);
                                              ❷
```

Returns `score and twenty` ❶

Returns `score` ❷

See Also

PHP documentation for `substr()` and for `strpos()` .

4.3 Replacing Part of a String

Problem

You want to replace just one part of a string with another string. For example, you want to obfuscate all but the last four digits of a phone number before printing it to the screen.

Solution

Use `substr_replace()` to replace a component of an existing string based on its position:

```
$string = '555-123-4567';
$replace = 'xxx-xxx'

$obfuscated = substr_replace($string, $replace, 0, strl
// xxx-xxx-4567
```

Discussion

PHP's `substr_replace()` function operates on a part of a string, similar to `substr()`, defined by an integer offset up to a specific length. Example 4-9 shows the full function signature.

Example 4-9. Full function signature of `substr_replace()`

```
function substr_replace(
    array|string $string,
    array|string $replace,
    array|int $offset,
    array|int|null $length = null
): string
```

Unlike its `substr()` analog, `substr_replace()` can operate either on individual strings or on collections of strings. If an array of strings is passed in with scalar values for `$replace` and `$offset`, the function will run the replacement on each string, as in Example 4-10.

Example 4-10. Replacing multiple substrings at once

```
$phones = [
    '555-555-5555',
    '555-123-1234',
    '555-991-9955'
];

$obfuscated = substr_replace($phones, 'xxx-xxx', 0, 7);

// xxx-xxx-5555
// xxx-xxx-1234
// xxx-xxx-9955
```

In general, developers have a lot of flexibility with the parameters in this function. Similar to `substr()`, the following are true:

- `$offset` can be negative, in which case replacements begin that number of characters from the *end* of the string.
- `$length` can be negative, representing the number of characters from the end of the string at which to stop replacing.

- If `$length` is `null`, it will internally become the same as the length of the input string itself.
- If `length` is `0`, `$replace` will be *inserted* into the string at the given `$offset`, and no replacement will take place at all.

Finally, if `$string` is provided as an array, all other parameters can be provided as arrays as well. Each element will represent a setting for the string in the same position in `$string`, as illustrated by Example 4-11.

Example 4-11. Replacing multiple substrings with array parameters

```
$phones = [
    '555-555-5555',
    '555-123-1234',
    '555-991-9955'
];

$offsets = [0, 0, 4];

$replace = [
    'xxx-xxx',
    'xxx-xxx',
    'xxx-xxxx'
];

$lengths = [7, 7, 8];

$obfuscated = substr_replace($phones, $replace, $offset

// xxx-xxx-5555
// xxx-xxx-1234
// 555-xxx-xxxx
```

NOTE

It is not a hard requirement for arrays passed in for `$string`, `$replace`, `$offset`, and `$length` to be all of the same size. PHP will not throw an error or warning if you pass arrays with different dimensions. Doing so will, however, result in unexpected output during the replacement operation—for example, truncating a string rather than replacing its contents. It's a good idea to validate that the dimensions of each of these four arrays all match.

The `substr_replace()` function is convenient if you know exactly *where* you need to replace characters within a string. In some situations, you might not know the position of a substring that needs to be replaced, but you want to instead replace occurrences of a *specific* substring. In those circumstances, you would want to use either `str_replace()` or `str_ireplace()`.

These two functions will search a specified string to find an occurrence (or many occurrences) of a specified substring and replace it with something else. The functions are identical in their call pattern, but the extra `i` in `str_ireplace()` indicates that it searches for a pattern in a *case-insensitive* fashion. Example 4-12 illustrates both functions in use.

Example 4-12. Searching and replacing within a string

```
$string = 'How much wood could a Woodchuck chuck if a w

$beaver = str_replace('woodchuck', 'beaver', $string);
                                                ❶

$ibeaver = str_ireplace('woodchuck', 'beaver', $string)
                                                ❷
```

How much wood could a Woodchuck chuck if a beaver could chuck wood? ❶

How much wood could a beaver chuck if a beaver could chuck wood? ❷

Both `str_replace()` and `str_ireplace()` accept an optional `$count` parameter that is passed by reference. If specified, this variable will be updated with the number of replacements the function performed. In Example 4-12, this return value would have been `1` and `2`, respectively, because of the capitalization of `Woodchuck`.

See Also

PHP documentation on `substr_replace()`, `str_replace()`, and `str_ireplace()`.

4.4 Processing a String One Byte at a

Time

Problem

You need to process a string of single-byte characters from beginning to end, one character at a time.

Solution

Loop through each character of the string as if it were an array. Example 4-13 will count the number of capital letters in a string.

Example 4-13. Count capital characters in a string

```
$capitals = 0;

$string = 'The Carriage held but just Ourselves - And I
for ($i = 0; $i < strlen($string); $i++) {
    if (ctype_upper($string[$i])) {
        $capitals += 1;
    }
}

// $capitals = 5
```

Discussion

Strings are not arrays in PHP, so you cannot loop over them directly. However, they do provide array-like access to individual characters within the string based on their position. You can reference individual characters by their integer offset (starting with 0), or even by a *negative* offset to start at the end of the string.

Array-like access isn't read-only, though. You can just as easily *replace* a single character in a string based on its position, as demonstrated by Example 4-14.

Example 4-14. Replacing a single character in a string

```
$string = 'A new recipe made my coffee stronger this mc
$string[31] = 'a';

// A new recipe made my coffee stranger this morning
```

It is also possible to convert a string *directly* to an array by using `str_split()` and then iterate over all items in the resulting array. This will work as an update to the Solution example, as illustrated in Example 4-15.

Example 4-15. Converting a string into an array directly

```
$capitals = 0;

$string = 'The Carriage held but just Ourselves - And I
$stringArray = str_split($string);
foreach ($stringArray as $char) {
    if (ctype_upper($char)) {
        $capitals += 1;
    }
}

// $capitals = 5
```

The downside of Example 4-15 is that PHP now has to maintain *two* copies of your data: the original string and the resultant array. This isn't a problem when handling small strings as in the example; if your strings instead represent entire files on disk, you will rapidly exhaust the memory available to PHP.

PHP makes accessing individual bytes (characters) within a string relatively easy without any changes in data type. Splitting a string into an array works but might be unnecessary unless you actually *need* an array of characters. Example 4-16 reimagines Example 4-15, using an array reduction technique rather than counting the capital letters in a string directly.

Example 4-16. Counting capital letters in a string with array reduction

```
$str = 'The Carriage held but just Ourselves - And Immc
```

```
$caps = array_reduce(str_split($str), fn($c, $i) => cty
```

NOTE

While Example 4-16 is functionally equivalent to Example 4-15, it is more concise and, consequently, more difficult to understand. While it is tempting to reimagine complex logic as one-line functions, unnecessary refactoring of your code for the sake of conciseness can be dangerous. The code might appear elegant but over time becomes more difficult to maintain.

The simplified reduction introduced in Example 4-16 is functionally accurate but still requires splitting the string into an array. It saves on lines of code in your program but still results in creating a second copy of your data. As mentioned before, if the strings over which you're iterating are large (e.g., massive binary files), this will rapidly consume the memory available to PHP.

See Also

PHP documentation on string access and modification, as well as documentation on `ctype_upper()`.

4.5 Generating Random Strings

Problem

You want to generate a string of random characters.

Solution

Use PHP's native `random_int()` function:

```
function random_string($length = 16)
{
    $characters = '0123456789abcdefghijklmnopqrstuvwxyz

    $string = '';
    while (strlen($string) < $length) {
```

```
            $string .= $characters[random_int(0, strlen($ch
        }
        return $string;
    }
```

Discussion

PHP has strong, *cryptographically secure* pseudorandom generator functions for both integers and bytes. It does not have a native function that generates random human-readable text, but the underlying functions can be used to create such a string of random text by leveraging lists of human-readable characters, as illustrated by the Solution example.

NOTE

A *cryptographically secure pseudorandom number generator* is a function that returns numbers with no distinguishable or predictable pattern. Even forensic analysis cannot distinguish between random noise and the output of a cryptographically secure generator.

A valid and potentially simpler method for producing random strings is to leverage PHP's `random_bytes()` function and encode the binary output as ASCII text. Example 4-17 illustrates two possible methods of using random bytes as a string.

Example 4-17. Creating a string of random bytes

```
$string = random_bytes(16);
```
❶

```
$hex = bin2hex($string);
```
❷

```
$base64 = base64_encode($string);
```
❸

❶ Because the string of binary bytes will be encoded in a different format, keep in mind that the number of bytes produced will *not* match the length of the final string.

❷

Encode the random string in hexadecimal format. Note that this format will double the length of the string—16 bytes is equivalent to 32 hexadecimal characters.

Leverage Base64 encoding to convert the raw bytes on readable characters. Note that this format increases the length of the string by 33%–36%.

See Also

PHP documentation on `random_int()` and on `random_bytes()`. Also Recipe 5.4 on generating random numbers.

4.6 Interpolating Variables Within a String

Problem

You want to include dynamic content in an otherwise static string.

Solution

Use double quotes to wrap the string and insert a variable, object property, or even function/method call directly in the string itself:

```
echo "There are {$_POST['cats']} cats and {$_POST['dogs
echo "Your username is {strlen($username)} characters l
echo "The car is painted {$car->color}.";
```

Discussion

Unlike single-quoted strings, double-quoted strings allow for complex, dynamic values as literals. Any word starting with a `$` character is interpreted as a variable name, unless that leading character is properly escaped.[2]

While the Solution example wraps dynamic content in curly braces, this is not a requirement in PHP. Simple variables can easily be written as is within a

double-quoted string and will be interpolated properly. However, more complex sequences become difficult to read without the braces. It's a highly recommended best practice to always enclose any value you want interpolated to make the string more readable.

Unfortunately, string interpolation has its limits. The Solution example illustrates pulling data out of the superglobal `$_POST` array and inserting it directly into a string. This is potentially dangerous, as that content is generated directly by the user, and the string could be leveraged in a sensitive way. In fact, string interpolation like this is one of the largest vectors for injection attacks against applications.

NOTE

In an injection attack, a third party can pass (or inject) executable or otherwise malicious input into your application and cause it to misbehave. More sophisticated ways to protect against this family of attacks are covered in Chapter 9.

To protect your string use against potentially malicious user-generated input, it's a good idea to instead use a format string via PHP's `sprintf()` function to filter the content. Example 4-18 rewrites part of the Solution example to protect against malicious `$_POST` data.

Example 4-18. Using format strings to produce an interpolated string

```
echo sprintf('There are %d cats and %d dogs.', $_POST['
```

Format strings are a basic form of input sanitization in PHP. In Example 4-18, you are explicitly assuming that the supplied `$_POST` data is numeric. The `%d` tokens within the format string will be replaced by the user-supplied data, but PHP will explicitly cast this data as integers during the replacement.

If, for example, this string were being inserted into a database, the formatting would protect against the potential of injection attacks against SQL interfaces. More complete methods of filtering and sanitizing user input are discussed in Chapter 9.

See Also

PHP documentation on variable parsing in double quotes and Heredoc as well as documentation on the `sprintf()` function.

4.7 Concatenating Multiple Strings Together

Problem

You need to create a new string from two smaller strings.

Solution

Use PHP's string concatenation operator:

```
$first = 'To be or not to be';
$second = 'That is the question';

$line = $first . ' ' . $second;
```

Discussion

PHP uses a single `.` character to join two strings together. This operator will also leverage type coercion to ensure that both values in the operation are strings before they're concatenated, as shown in Example 4-19.

Example 4-19. String concatenation

```
print 'String ' . 2;
                          ❶

print 2 . ' number';
                          ❷

print 'Boolean ' . true;
                          ❸

print 2 . 3;
                          ❹
```

Prints `String 2` ❶

Prints `2 number` ❷

Prints `Boolean 1` because Boolean values are cast to integers and then to strings ❸

Prints `23` ❹

The string concatenation operator is a quick way to combine simple strings, but it can become somewhat verbose if you use it to combine multiple strings with whitespace. Consider Example 4-20, where you try to combine a list of words into a string, each separated by a space.

Example 4-20. Verbosity in concatenating large groups of strings

```
$words = [
    'Whose',
    'woods',
    'these',
    'are',
    'I',
    'think',
    'I',
    'know'
];

$option1 = $words[0] . ' ' . $words[1] . ' ' . $words[2
         ' ' . $words[4] . ' ' . $words[5] . ' ' . $wor
         ' ' . $words[7];                                ❶


$option2 = '';
foreach ($words as $word) {
    $option2 .= ' ' . $word;                             ❷

}
$option2 = ltrim($option2);                              ❸
```

❶ One option is to individually concatenate each word in the collection with whitespace separators. As the word list grows, this quickly becomes unwieldy.

❷

You can, instead, loop over the collection and build up a concatenated string without accessing each item in the collection individually.

When using a loop, you might end up with unnecessary whitespace. You need to remember to trim extraneous spaces from the start of the string.

Large, repetitive concatenation routines can be replaced by native PHP functions like `implode()`. This function in particular accepts an array of data to be joined and a definition of the character (or characters) to be used between data elements. It returns the final, concatenated string.

NOTE

Some developers prefer to use `join()` instead of `implode()` as it's seen to be a more descriptive name for the operation. The fact is, `join()` is an alias of `implode()`, and the PHP compiler doesn't care which you use.

Rewriting Example 4-20 to use `implode()` makes the entire operation much simpler, as demonstrated by Example 4-21.

Example 4-21. A concise approach to string concatenation

```
$words = [
    'Whose',
    'woods',
    'these',
    'are',
    'I',
    'think',
    'I',
    'know'
];

$string = implode(' ', $words);
```

Take care to remember the parameter order for `implode()`. The string separator comes *first*, followed by the array over which you want to iterate. Earlier versions of PHP (prior to version 8.0) allowed the parameters to be specified in the opposite order. This behavior (specifying the array first and

the separator second) was deprecated in PHP 7.4. As of PHP 8.0, this will throw a `TypeError`.

If you're using a library written prior to PHP 8.0, be sure you test that it's not misusing either `implode()` or `join()` before you ship your project to production.

See Also

PHP documentation on `implode()`.

4.8 Managing Binary Data Stored in Strings

Problem

You want to encode data directly as binary rather than as an ASCII-formatted representation, or you want to read data into your application that was explicitly encoded as binary data.

Solution

Use `unpack()` to extract binary data from a string:

```
$unpacked = unpack('S1', 'Hi'); // [1 => 26952]
```

Use `pack()` to write binary data to a string:

```
$packed = pack('S13', 72, 101, 108, 108, 111, 44, 32, 1
              114, 108, 100, 33); // 'Hello, world!'
```

Discussion

Both `pack()` and `unpack()` empower you to operate on raw binary strings, assuming you know the format of the binary string you're working

with. The first parameter of each function is a format specification. This specification is determined by specific format codes, as defined in Table 4-2.

Table 4-2. Binary format string codes

Code	Description
`a`	Null-padded string
`A`	Space-padded string
`h`	Hex string, low nibble first
`H`	Hex string, high nibble first
`c`	Signed char
`C`	Unsigned char
`s`	Signed short (always 16-bit, machine byte order)
`S`	Unsigned short (always 16-bit, machine byte order)
`n`	Unsigned short (always 16-bit, big-endian byte order)
`v`	Unsigned short (always 16-bit, little-endian byte order)
`i`	Signed integer (machine-dependent size and byte order)
`I`	Unsigned integer (machine-dependent size and byte order)
`l`	Signed long (always 32-bit, machine byte order)
`L`	Unsigned long (always 32-bit, machine byte order)
`N`	Unsigned long (always 32-bit, big-endian byte order)
`V`	Unsigned long (always 32-bit, little-endian byte order)

Code	Description
`q`	Signed long long (always 64-bit, machine byte order)
`Q`	Unsigned long long (always 64-bit, machine byte order)
`J`	Unsigned long long (always 64-bit, big-endian byte order)
`P`	Unsigned long long (always 64-bit, little-endian byte order)
`f`	Float (machine-dependent size and representation)
`g`	Float (machine-dependent size, little-endian byte order)
`G`	Float (machine-dependent size, big-endian byte order)
`d`	Double (machine-dependent size and representation)
`e`	Double (machine-dependent size, little-endian byte order)
`E`	Double (machine-dependent size, big-endian byte order)
`x`	Null byte
`X`	Back up one byte
`Z`	Null-padded string
`@`	Null-fill to absolute position

When defining a format string, you can specify each byte type individually or leverage an optional repeating character. In the Solution examples, the number of bytes is explicitly specified with an integer. You could just as easily use an asterisk (`*`) to specify that a type of byte repeats through the end of the string as follows:

```
$unpacked = unpack('S*', 'Hi'); // [1 => 26952]
$packed = pack('S*', 72, 101, 108, 108, 111, 44, 32, 11
```

```
                    114, 108, 100, 33); // 'Hello, world!'
```

PHP's ability to convert between byte encoding types via `unpack()` also provides a simple method of converting ASCII characters to and from their binary equivalent. The `ord()` function will return the value of a specific character, but it requires looping over each character in a string if you want to unpack each in turn, as demonstrated in Example 4-22.

Example 4-22. Retrieving character values with `ord()`

```
$ascii = 'PHP Cookbook';

$chars = [];
for ($i = 0; $i < strlen($ascii); $i++) {
    $chars[] = ord($ascii[$i]);
}

var_dump($chars);
```

Thanks to `unpack()`, you don't need to explicitly iterate over the characters in a string. The `c` format character references a signed character, and `C` a signed one. Rather than building a loop, you can leverage `unpack()` directly as follows to get an equivalent result:

```
$ascii = 'PHP Cookbook';
$chars = unpack('C*', $ascii);

var_dump($chars);
```

Both the preceding `unpack()` example and the original loop implementation in Example 4-22 produce the following array:

```
array(12) {
  [1]=>
  int(80)
  [2]=>
  int(72)
  [3]=>
  int(80)
  [4]=>
  int(32)
```

```
    [5]=>
    int(67)
    [6]=>
    int(111)
    [7]=>
    int(111)
    [8]=>
    int(107)
    [9]=>
    int(98)
    [10]=>
    int(111)
    [11]=>
    int(111)
    [12]=>
    int(107)
}
```

See Also

PHP documentation on `pack()` and `unpack()`.

[1] This string is a byte representation, formatted in octal notation, of “Hello World!”

[2] Review Table 4-1 for more on double-character escape sequences.