CAPÍTULO 3

Introducción a PHP

En el Capítulo1, expliqué que PHP es el lenguaje que usamos para hacer que el servidor genere una salida dinámica, salida que es potencialmente diferente cada vez que un navegador solicita una página. En este capítulo, comenzaremos a aprender este sencillo pero potente lenguaje. Durante los siguientes capítulos, hasta el Capítulo 7, trataremos este tema.

Te animo a que desarrolles tu código PHP con uno de los EDI de la lista del Capítulo 2. Te ayudará a detectar errores tipográficos y a acelerar enormemente el aprendizaje si lo comparamos con un editor con menos funciones.

Muchos de estos entornos de desarrollo te permitirán ejecutar el código PHP y ver los resultados que se comentan en este capítulo. También mostraré cómo hay que integrar PHP en un archivo HTML para que puedas apreciar cómo se ve el resultado en una página web (la forma en la que los usuarios la verán en última instancia). Pero ese paso, por muy emocionante que sea al principio, no es realmente importante en esta etapa.

En producción, tus páginas web serán una combinación de PHP, HTML, JavaScript y algunas sentencias MySQL, y se diseñarán con CSS. Además, cada página puede llevar a otras páginas, lo que porporciona a los usuarios los medios para hacer clic en los enlaces y rellenar formularios. Nosotros podemos eludir toda esa complejidad mientras aprendemos cada uno de los lenguajes. Por ahora concéntrate en escribir código PHP de manera que obtengas el resultado que esperas, ¡o al menos que entiendas el resultado que realmente obtienes!

Inclusión de PHP en HTML

Por defecto, los documentos PHP terminan con la extensión *.php*. Cuando un servidor web encuentra esta extensión en un archivo que se le ha solicitado, lo pasa automáticamente al procesador PHP. Por supuesto, los servidores web son altamente configurables, y algunos desarrolladores web obligan a que los archivos terminen en *.htm* o *.html* para que también los analice el procesador PHP, generalmente porque quieren ocultar el uso de PHP.

El programa PHP es responsable de devolver un archivo limpio, adecuado para que lo muestre un navegador web. En su forma más sencilla, un documento PHP solo producirá HTML. Para comprobarlo puedes elegir cualquier documento HTML normal y guardarlo

como un documento PHP (por ejemplo, puedes guardar *index.html* como *index.php*), y se mostrará idéntico al original.

Para activar los comandos PHP, necesitas aprender una nueva etiqueta. Aquí está la primera parte:

```
<?php
```

Lo primero que observamos es que la etiqueta no se ha cerrado. Y no se ha hecho porque a continuación se colocan secciones enteras de PHP debajo de esta etiqueta, que terminarán solo cuando se encuentren la parte de cierre, que es la siguiente:

```
?>
```

Un pequeño programa PHP "Hello World" podría parecerse al del Ejemplo 3-1.

Ejemplo 3-1. Llamada de PHP

```
<?php
  echo "Hello world";
?>
```

Esta etiqueta admite un uso bastante flexible. Algunos programadores abren la etiqueta al principio de un documento y la cierran al final, y dan salida a cualquier HTML directamente desde los comandos PHP. Otros, sin embargo, optan por insertar solo fragmentos lo más pequeños posible de PHP dentro de estas etiquetas dondequiera que se requieran secuencias de comandos dinámicas, y dejan el resto del documento en HTML estándar.

El segundo tipo de programador generalmente argumenta que su estilo de codificación proporciona una ejecución más rápida del código, mientras que el primero dice que el aumento de velocidad es tan mínimo que no justifica la complejidad adicional de entrar y salir de PHP muchas veces en un solo documento.

A medida que vayas aprendiendo, seguramente descubrirás tu estilo preferido de desarrollo PHP, pero para que los ejemplos de este libro sean más fáciles de seguir, he adoptado el enfoque de mantener el mínimo número de transferencias entre PHP y HTML, que generalmente será de solo una o dos veces en un documento.

Por cierto, hay una ligera variación en la sintaxis de PHP. Si navegamos por Internet en busca de ejemplos de PHP, también podemos encontrar el código donde la sintaxis de apertura y cierre se parece a esto:

```
<?
  echo "Hello world";
?>
```

Aunque no es tan obvio que se esté llamando al analizador de PHP, esta es una sintaxis alternativa válida que también suele funcionar. Pero desaconsejo su uso, ya que es incompatible con XML y ahora está obsoleta (lo que significa que ya no se recomienda su uso y podría no tener soporte en versiones futuras).

Si solo existe código PHP en un archivo, puedes omitir el cierre ?>. Esto puede ser una buena práctica, ya que asegurará que no haya exceso de espacios en blanco que se filtren de nuestros archivos PHP (especialmente importante cuando estamos escribiendo código orientado a objetos).

Ejemplos de este libro

Los ejemplos de este libro se han archivado en *www.marcombo.info*, ahorrándote así el tiempo que tendrías que dedicar a escribirlos. Puedes descargar el archivo en tu ordenador siguiendo los pasos de la primera página del libro.

Existe también una página web para este libro, en la que aparecen listados de erratas, ejemplos, y cualquier información adicional. Puedes acceder a esta página en *http://bit.ly/lpmjch_5e*.

Además de listar los ejemplos por el número de capítulo y ejemplo (como *example3-1.php*), el archivo también contiene un directorio adicional llamado *named_examples* en el que encontrarás todos los ejemplos que te sugiero que guardes con un nombre de archivo específico (como el del Ejemplo 3-4, que debería guardarse como *test1.php*).

Estructura de PHP

Vamos a avanzar mucho en esta sección. No es demasiado difícil, pero yo recomiendo que lo hagas con cuidado, ya que sienta las bases para comprender el resto del contenido de este libro. Como siempre, hay algunas preguntas útiles al final del capítulo con las que puedes comprobar cuánto has aprendido.

Utilización de comentarios

Hay dos maneras de añadir comentarios al código PHP. La primera convierte una línea en un comentario si se le anteponen un par de barras inclinadas:

```
// This is a comment
```

Esta versión de la función de comentarios es una excelente manera de eliminar temporalmente una línea de código de un programa que nos está dando errores. Por ejemplo, podrías usar dicho comentario para ocultar una línea de código en el proceso de depuración hasta que la necesites, escribiendo:

```
// echo "X equals $x";
```

También puedes utilizar este tipo de comentario directamente después de una línea de código para describir lo que hace, así:

```
$x += 10; // Increment $x by 10
```

Cuando necesites hacer comentarios de varias líneas, hay un segundo tipo de comentario, como el del Ejemplo 3-2.

Ejemplo 3-2. Un comentario de varias líneas

```
<?php
/* This is a section
   of multiline comments
   which will not be
   interpreted */
?>
```

Puedes utilizar los pares de caracteres /* y */ para abrir y cerrar comentarios casi en cualquier lugar que desees dentro del código. La mayoría de los programadores, si no todos, utilizan este constructor para comentar provisionalmente secciones de código que no funcionan o que, por una razón u otra, no desean que el intérprete actúe sobre ellas.

Un error muy habitual es usar /* y */ para comentar una gran sección de código que ya contiene una sección con comentarios que usa esos caracteres. No se pueden anidar comentarios de esta manera, ya que el intérprete PHP no sabrá dónde termina un comentario y mostrará un mensaje de error. Sin embargo, si utilizamos un editor de programas o EDI con resaltado de sintaxis, este tipo de error es más fácil de detectar.

Sintaxis básica

PHP es un lenguaje bastante sencillo, que tiene sus orígenes en C y Perl, pero se parece más a Java. También es muy flexible, pero hay algunas reglas que necesitas aprender sobre su sintaxis y estructura.

Punto y coma

Seguramente has observado que en los ejemplos anteriores los comandos PHP se cerraban con un punto y coma, como este:

```
$x += 10;
```

Probablemente la causa más frecuente de errores que encontrarás en PHP es olvidar este punto y coma. Esto hace que PHP trate múltiples sentencias como una sola y no las pueda entender, y muestre un mensaje `Parse error` (error de análisis).

El símbolo $

El símbolo $ se ha llegado a utilizar de muchas maneras distintas en diferentes lenguajes de programación. Por ejemplo, en BASIC, los nombres de variables terminaban con este símbolo para denotarlas como cadenas.

En PHP, sin embargo, debes colocar un $ delante de todas las variables. Esto es necesario para que el analizador de PHP actúe más rápido, ya que sabe instantáneamente cuando se encuentra con una variable. Independientemente de que las variables sean números, cadenas o matrices, todas se deben parecer algo a las del Ejemplo 3-3.

Ejemplo 3-3. Tres clases diferentes de asignación de variables

```
<?php
  $mycounter = 1;
  $mystring  = "Hello";
  $myarray   = array("One", "Two", "Three");
?>
```

Y realmente esa es toda la sintaxis que tienes que recordar. A diferencia de lenguajes como Python, que son muy estrictos sobre cómo sangrar y diseñar el código, PHP te deja completa libertad para usar (o no usar) todo el sangrado y espaciado que desees. De hecho, se fomenta el uso sensato de los espacios en blanco (junto con los detallados comentarios) para ayudarte a entender el código cuando tengas que volver a él de nuevo. También ayuda a otros programadores cuando tienen que mantener tu código.

Variables

Hay un símil que te ayudará a entender de qué tratan las variables PHP. ¡Piensa en ellas como pequeñas (o grandes) cajas de cerillas! Son cajas de cerillas que has pintado y en las que has escrito nombres.

Variables de cadena

Imagina que tienes una caja de cerillas en la que has escrito la palabra *username* (nombre de usuario). Luego escribes *Fred Smith* en un pedazo de papel y lo colocas en la caja (ver la Figura 3-1). Este es el mismo proceso que se sigue para asignar un valor de cadena a una variable, así:

```
$username = "Fred Smith";
```

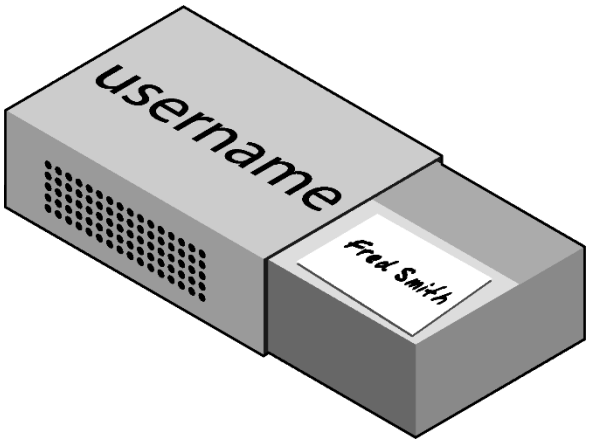

Figura 3-1. Puedes ver las variables como cajas de cerillas que contienen elementos

Las comillas indican que "Fred Smith" es una *string* (cadena de caracteres). Debes

encerrar cada cadena entre comillas o apóstrofes (comillas simples), aunque hay una sutil diferencia entre los dos tipos de comillas, que se explica más adelante. Cuando quieras ver lo que hay en la caja, ábrela, saca el trozo de papel y léelo. En PHP, esto se hace así (muestra el contenido de la variable en pantalla):

```
echo $username;
```

O puedes asignar el contenido a otra variable (fotocopiar el papel y colocar la copia en otra caja de cerillas), así:

```
$current_user = $username;
```

Si deseas empezar a probar PHP, puedes introducir los ejemplos de este capítulo en un EDI (como se recomienda al final del Capítulo 2) para ver resultados inmediatos, o puedes introducir el código del Ejemplo 3-4 en un editor de programas (también expuesto en el Capítulo 2) y guardarlo en la carpeta principal de tu servidor como *test1.php*.

Ejemplo 3-4. Tu primer programa PHP

```
<?php // test1.php
  $username = "Fred Smith";
  echo $username;
  echo "<br>";
  $current_user = $username;
  echo $current_user;
?>
```

Ahora puedes llamarlo introduciendo lo siguiente en la barra de direcciones del navegador:

```
http://localhost/test1.php
```

En el improbable caso de que durante la instalación de tu servidor web (como se detalla en el Capítulo 2) hayas cambiado el puerto asignado al servidor a cualquier otro que no sea 80, entonces debes colocar ese número de puerto dentro del URL en este y en todos los demás ejemplos de este libro. Así, por ejemplo, si cambiaras el puerto a 8080, el URL anterior se convertiría en:

```
http://localhost:8080/test1.php
```

No volveré a hacer referencia a esto, así que recuerda usar el número de puerto (si es necesario) cuando intentes reproducir los ejemplos o escribas tu propio código.

El resultado de ejecutar este código es que debería aparecer dos veces el nombre Fred Smith, el primero de los cuales es el resultado del comando `echo $username` y el segundo del comando `echo $current_user`.

Variables numéricas

Las variables no solo pueden contener cadenas de caracteres, también pueden contener números. Si volvemos a la analogía de la caja de cerillas, para almacenar el número 17

en la variable `$count`, el equivalente sería colocar, digamos, 17 cuentas en una caja de cerillas en la que se ha escrito la palabra *count* (cuenta):

```
$count = 17;
```

También puedes utilizar un número en punto flotante (que contenga un punto decimal). La sintaxis es la misma:

```
$count = 17.5;
```

Para ver el contenido de la caja de cerillas, basta con abrirla y contar las cuentas. En PHP, asignaríamos el valor de `$count` a otra variable o quizás solo haríamos echo (presentación en pantalla) en el navegador web.

Matrices

¿Qué son las matrices? Bueno, puedes pensar en ellas como varias cajas de cerillas pegadas entre sí. Por ejemplo, supongamos que queremos almacenar los nombres de los jugadores de un equipo de fútbol de cinco personas en una matriz llamada `$team`. Para ello, podríamos pegar cinco cajas de cerillas una al lado de la otra y escribir los nombres de cada uno de los jugadores en trozos separados de papel, y colocar uno en cada caja de cerillas.

En la parte superior del ensamblaje de la caja de cerillas escribiríamos la palabra *team* (equipo) (ver Figura 3-2). El equivalente de esto en PHP sería lo siguiente:

```
$team = array('Bill', 'Mary', 'Mike', 'Chris', 'Anne');
```

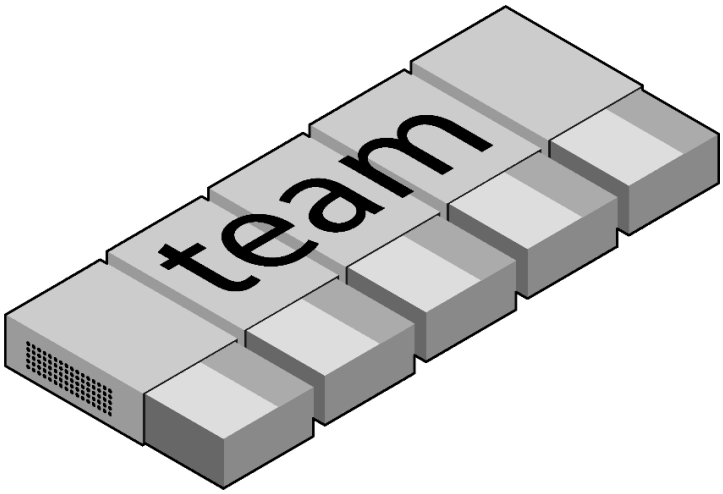

Figura 3-2. Una matriz es como varias cajas de cerillas pegadas unas a otras

Esta sintaxis es más complicada que la de los otros ejemplos que hemos visto hasta ahora. El código para crear la matriz está compuesto por el siguiente constructor:

```
array();
```

y contiene cinco cadenas. Cada cadena está encerrada en apóstrofes, y las cadenas deben estar separadas por comas.

Si entonces quisiéramos saber quién es el jugador 4, podríamos usar este comando:

```
echo $team[3]; // Displays the name Chris
```

La razón por la que la declaración anterior tiene el número 3 y no el 4, es porque el primer elemento de una matriz PHP es en realidad el elemento cero, por lo que los números de jugador van de 0 a 4.

Matrices de dos dimensiones

Hay muchas más cosas que puedes hacer con las matrices. Por ejemplo, en lugar de tratarse de una hilera unidimensional de cajas de cerillas, pueden ser matrices bidimensionales o pueden incluso tener más dimensiones.

Como ejemplo de una matriz bidimensional, digamos que queremos seguir el rastro de un juego de tres en raya, que requiere una estructura de datos de nueve celdas dispuestas en un cuadrado de 3 × 3. Para representar esto con cajas de cerillas, imagínate nueve de ellas pegadas unas a otras formando una matriz de tres filas por tres columnas (ver Figura 3-3).

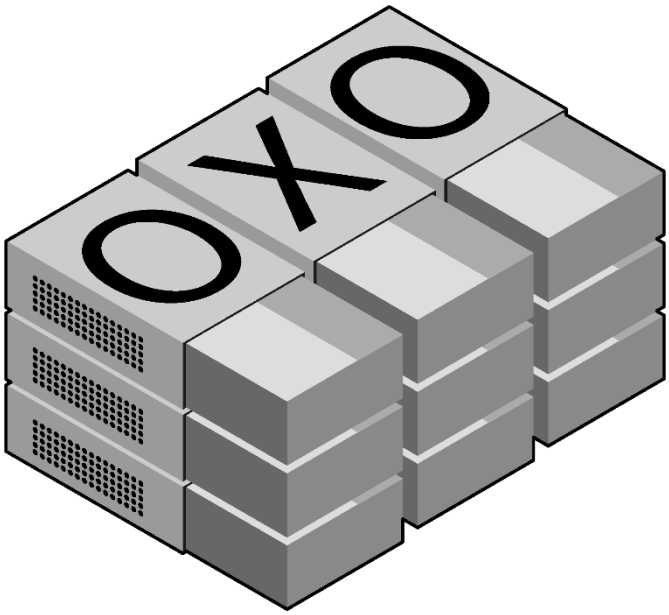

Figura 3-3. Matriz multidimensional simulada con cajas de cerillas

Ahora puedes colocar una hoja de papel con una *x* o una *o* en la caja de cerillas adecuada para cada jugada. Para hacer esto en código PHP, tienes que configurar una matriz que contenga tres matrices más, como en el Ejemplo 3-5, en el que la matriz se configura con la partida que se está jugando.

Ejemplo 3-5. Definición de una matriz bidimensional

```
<?php
  $oxo = array(array('x', ' ', 'o'),
               array('o', 'o', 'x'),
               array('x', 'o', ' '));
?>
```

De nuevo, hemos avanzado un paso en complejidad, pero es fácil de entender si captas la sintaxis básica de la matriz. Hay tres constructores de `array()` anidadas dentro del constructor externo `array()`. Con cada jugada, hemos rellenado cada fila con una matriz que consta solo de un carácter: una *x*, una *o*, o un espacio en blanco. (Usamos un espacio en blanco para que todas las celdas tengan la misma anchura cuando se visualizan).

Para conocer el tercer elemento en la segunda fila de esta matriz, usaríamos el siguiente comando PHP, que mostrará una `x`:

```
echo $oxo[1][2];
```

Recuerda que los índices de las matrices (punteros de los elementos dentro de una matriz) comienzan en cero, no en uno, por lo que `[1]` en el comando anterior se refiere a la segunda de las tres matrices, y `[2]` se refiere a la tercera posición dentro de esa matriz. Este comando nos devolverá el contenido de la caja de cerillas situada en la intersección de la tercera columna y la segunda fila.

Como se mencionó anteriormente, podemos tratar matrices con más dimensiones aún, simplemente creando matrices dentro de otras matrices. Sin embargo, en este libro no trataremos con matrices de más de dos dimensiones.

Y no te preocupes si todavía tienes dificultades para enfrentarte al uso de matrices, ya que el tema se explica con detalle en el Capítulo 6.

Reglas de denominación de variables

Al crear variables PHP, debemos seguir estas cuatro reglas:

- Los nombres de las variables, después del símbolo del dólar, deben comenzar con una letra del alfabeto o el carácter _ (subrayado).
- Los nombres de variables solo pueden contener los caracteres `a-z`, `A-Z`, `0-9`, y `_` (subrayado).
- Los nombres de las variables no pueden contener espacios. Si un nombre de variable debe incluir más que una palabra, una buena idea es separar las palabras con el carácter _ (subrayado) (p. ej., `$user_name`).
- Los nombres de las variables distinguen entre mayúsculas y minúsculas. La variable `$High_Score` no es la misma que la variable `$high_score`.

Para permitir caracteres ASCII extendidos que incluyan acentos, PHP también admite los bytes de 127 a 255 en nombres de variables. Pero a menos que tu código lo vayan a mantener programadores acostumbrados a esos caracteres, probablemente sea mejor evitarlos, porque los programadores que usan teclados en inglés tendrán dificultades para acceder a ellos.

Operadores

Los *operadores* permiten especificar operaciones matemáticas, como sumas, restas, multiplicaciones y divisiones. Pero también existen otros tipos de operadores, como la cadena de caracteres, la comparación y los operadores lógicos. Las matemáticas en PHP se parecen mucho a la aritmética básica. Por ejemplo, la siguiente declaración da como resultado 8:

```
echo 6 + 2;
```

Antes de empezar a saber lo que PHP puede hacer por nosotros, dedicaremos un momento a aprender acerca de los diferentes operadores que podemos utilizar.

Operadores aritméticos

Los operadores aritméticos hacen lo que esperaríamos de ellos: realizan cálculos matemáticos. Podemos usarlos para las cuatro operaciones principales (sumar, restar, multiplicar y dividir) así como para encontrar un módulo (el resto después de una división) y para incrementar o decrementar un valor (ver la Tabla 3-1).

Tabla 3-1. Operadores aritméticos

Operador	Descripción	Ejemplo
+	Suma	$j + 1
-	Resta	$j - 6
*	Multiplicación	$j * 11
/	División	$j / 4
%	Módulo (resto después de una división)	$j % 9
++	Incremento	++$j
--	Decremento	--$j
**	Exponenciación (o potencia)	$j**2

Operadores de asignación

Estos operadores asignan valores a variables. Comienzan con el más sencillo = y pasan a +=, -=, etc. (ver la Tabla 3-2). El operador += suma el valor de la derecha al valor de la variable de la izquierda, en lugar de reemplazar totalmente el valor de la izquierda. Por lo tanto, si `$count` comienza con el valor 5, la expresión:

```
$count += 1;
```

pone `$count` a 6, al igual que la declaración de asignación más familiar:

```
$count = $count + 1;
```

Tabla 3-2. Operadores de asignación

Operador	Ejemplo	Equivalente a
=	$j = 15	$j = 15
+=	$j += 5	$j = $j + 5
-=	$j -= 3	$j = $j - 3
*=	$j *= 8	$j = $j * 8
/=	$j /= 16	$j = $j / 16
.=	$j .= $k	$j = $j . $k
%=	$j %= 4	$j = $j % 4

Operadores de comparación

Los operadores de comparación se utilizan generalmente dentro de un constructor, como por ejemplo, una declaración `if` en la que se deben comparar dos elementos. Por ejemplo, es posible que quieras saber si una variable cuyo valor se ha estado incrementando ha alcanzado un valor específico, o bien si el valor de otra variable es menor que un valor establecido, etc. (ver la Tabla 3-3).

Tabla 3-3. Operadores de comparación

Operador	Descripción	Ejemplo
==	es igual a	$j == 4
!=	no es igual a	$j != 21
>	es mayor que	$j > 3
<	es menor que	$j < 100
>=	es mayor que o igual a	$j >= 15
<=	es menor que o igual a	$j <= 8
<>	no es igual a	$j <> 23
===	es idéntico a	$j === "987"
!==	no es idéntico a	$j !== "1.2e3"

Observa la diferencia entre = y ==. El primero es un operador de asignación y el segundo es un operador relacional. Incluso los programadores avanzados pueden a veces confundirlos cuando codifican apresuradamente, así que ten cuidado.

Operadores lógicos

Si no los has usado antes, los operadores lógicos pueden ser un poco desalentadores al principio. Pero piensa en ellos de la misma forma en la que usarías la lógica en español. Por ejemplo, podrías decirte a ti mismo: "Si es más tarde de las 12 p. m. y es antes de las 2 p. m., almorzar". En PHP, el código para esto podría ser algo así como lo siguiente (usando el tiempo militar):

```
if ($hour > 12 && $hour < 14) dolunch();
```

Aquí hemos trasladado el conjunto de instrucciones para ir a almorzar a una función que tendremos que crear más tarde llamada `dolunch`.

Como muestra el ejemplo anterior, generalmente se utiliza un operador lógico para combinar los resultados de dos de los operadores de comparación mostrados en la sección anterior. Un operador lógico también puede ser la entrada a otro operador lógico: "Si la hora es posterior a las 12 p. m. y anterior a las 2 p. m., o si el olor de un asado está presente en el pasillo y hay platos en la mesa". Como regla general, si algo tiene un valor `TRUE` (VERDADERO) o `FALSE` (FALSO), lo puede tratar un operador lógico. Un operador lógico recibe dos entradas verdaderas o falsas y produce un resultado verdadero o falso.

La Tabla 3-4 muestra los operadores lógicos.

Tabla 3-4. Operadores lógicos

Operador	Descripción	Ejemplo
`&&`	And (Y)	`$j == 3 && $k == 2`
`and`	Baja prioridad and (y)	`$j == 3 and $k == 2`
`||`	Or (O)	`$j < 5 || $j > 10`
`or`	Baja prioridad or (o)	`$j < 5 or $j > 10`
`!`	Not (No)	`! ($j == $k)`
`xor`	Or exclusivo	`$j xor $k`

Observa que `&&` se puede intercambiar en general con `and` (y); lo mismo es cierto para `||` y `or` (o). Sin embargo, debido a que `and` y `or` tienen una prioridad menor, debemos evitar usarlos excepto cuando son la única opción, como en la siguiente declaración, en la cual *debemos* usar el operador `or` (`||` no se puede usar para forzar la ejecución de una segunda declaración si la primera falla):

```
$html = file_get_contents($site) or die("Cannot download from
$site");
```

El operador que menos se utiliza de todos ellos es `xor`, que significa *or exclusivo* y devuelve un valor `TRUE` si cualquiera de los valores es `TRUE`, pero devuelve un valor `FALSE` si ambas entradas son `TRUE` o ambas entradas son `FALSE`. Para entender esto, imagínate que quieres crear tu propio limpiador con artículos del hogar. El amoníaco es un buen limpiador, y también lo es la lejía, así que quieres que el limpiador tenga uno de estos. Pero el limpiador no debe tener ambos, porque la combinación es peligrosa. En PHP, podemos representar esto de la siguiente manera:

```
$ingredient = $ammonia xor $bleach;
```

En este ejemplo, si `$ammonia` o `$bleach` es `TRUE`, `$ingredient` también será `TRUE`. Pero si ambos son `TRUE` o ambos son `FALSE`, `$ingredient` será `FALSE`.

Asignación de valores a variables

La sintaxis para asignar un valor a una variable es siempre `variable = value` *(variable = valor)*. O, para reasignar el valor a otra variable, es `other_variable = variable` *(otra variable = variable)*.

También hay otro par de operadores de asignación que te resultarán útiles. Por ejemplo, ya hemos visto esto:

```
$x += 10;
```

que le dice al analizador de PHP que agregue el valor a la derecha (en este caso, el valor 10) a la variable `$x`. Asimismo, podríamos restarlo haciendo lo siguiente:

```
$y -= 10;
```

Incremento y decremento de una variable

Sumar o restar 1 es una operación tan corriente que PHP proporciona operadores especiales para ello. Puedes utilizar una de las siguientes opciones en lugar de los operadores `+=` y `-=`:

```
++$x;
--$y;
```

Junto con una prueba (una declaración `if`), podemos utilizar el siguiente código:

```
if (++$x == 10) echo $x;
```

Esto le dice a PHP que *primero* aumente el valor de `$x` y luego pruebe si tiene el valor 10 y, si lo tiene, que presente el resultado. Pero también puedes pedirle a PHP que incremente (o, como en el siguiente ejemplo, decremente) una variable *después* de haber probado su valor, así:

```
if ($y-- == 0) echo $y;
```

lo que da un resultado ligeramente diferente. Supongamos que `$y` comienza en `0` antes de que se ejecute la declaración. La comparación devolverá el resultado `TRUE`, pero `$y` se pondrá a `-1` después de que se haya hecho la comparación. Entonces, ¿qué mostrará la declaración `echo`? ¿`0` o `-1`? Trata de adivinarlo, y luego prueba la declaración en un procesador PHP para confirmarlo. Debido a que esta combinación de declaraciones es confusa, se debe tomar como un ejemplo educativo y no como una guía de buen estilo de programación.

En resumen, una variable se incrementa o decrementa antes de la prueba si el operador se coloca antes de la variable, mientras que la variable se incrementa o decrementa después de la prueba si el operador está situado después de la variable.

Por cierto, la respuesta correcta a la pregunta anterior es que la declaración `echo` muestra el resultado `-1`, porque `$y` se decrementó justo después de que se accediera a él en la declaración `if`, y antes de la declaración `echo`.

Concatenación de cadenas de caracteres

Concatenación es un término algo arcano que significa poner algo después de otra cosa. Así, la concatenación de cadenas utiliza el punto (.) para añadir una cadena de caracteres a otra. La forma más sencilla de hacerlo es la siguiente manera:

```
echo "You have " . $msgs . " messages.";
```

Si suponemos que la variable `$msgs` tiene el valor de `5`, la salida de esta línea de código será la siguiente:

```
You have 5 messages.
```

Del mismo modo que se puede añadir un valor a una variable numérica con el operador `+=`, se puede añadir una cadena a otra mediante `.=`, así:

```
$bulletin .= $newsflash;
```

En este caso, si `$bulletin` contiene un boletín de noticias y `$newsflash` tiene un flash de noticias, el comando agrega el flash de noticias al boletín de noticias para que `$bulletin` ahora incluya ambas cadenas de texto.

Tipos de cadenas

PHP admite dos tipos de cadenas que se denotan por el tipo de comillas que usemos. Si deseamos asignar una cadena literal y preservar el contenido exacto, debemos utilizar comillas simples (apóstrofes), así:

```
$info = 'Preface variables with a $ like this: $variable';
```

En este caso, cada carácter dentro de la cadena entre comillas simples se asigna a `$info`. Si hubiéramos usado comillas dobles, PHP habría intentado evaluar `$variable` como una variable.

Por otro lado, cuando se desea incluir el valor de una variable dentro de una cadena, se hace mediante cadenas con comillas dobles:

```
echo "This week $count people have viewed your profile";
```

Como puedes observar, esta sintaxis también ofrece una opción más simple para la concatenación en la que no necesitas usar un punto, o cerrar y reabrir comillas, para agregar una cadena a otra. Esto se llama *sustitución de variables*, y algunos programadores la usan muy a menudo, mientras que otros no la usan en absoluto.

Caracteres de escape

A veces una cadena contiene necesariamente caracteres con significados especiales que se podrían interpretar de forma incorrecta. Por ejemplo, la siguiente línea de código no funcionará, porque la segunda comilla que se encuentra en la ortografía de la palabra le dirá al analizador de PHP que se ha alcanzado el final de la cadena. En consecuencia, el resto de la línea se rechazará por error:

```
$text = 'My spelling's atroshus'; // Erroneous syntax
```

Para corregir esto, podemos añadir una barra invertida justo antes de la comilla para decirle a PHP que trate el carácter de manera literal y que no lo interprete:

```
$text = 'My spelling\'s still atroshus';
```

Podemos realizar este truco en casi todas las situaciones en las que PHP, de no ser así, devolvería un error al intentar interpretar un carácter. Por ejemplo, la siguiente cadena con comillas dobles se asignará correctamente:

```
$text = "She wrote upon it, \"Return to sender\".";
```

Además, podemos utilizar caracteres de escape para insertar varios caracteres especiales en cadenas, como tabuladores, saltos de línea y retornos de carro. Estos están representados, como puedes adivinar, por `\t`, `\n`, y `\r`. He aquí un ejemplo que utiliza el tabulador para diseñar un encabezado. Se incluye aquí solo para ilustrar los escapes, ya que en las páginas web siempre hay mejores maneras de realizar el diseño:

```
$heading = "Date\tName\tPayment";
```

Estos caracteres especiales precedidos por la barra invertida solo funcionan en cadenas entre comillas dobles. En cadenas entre comillas, la cadena anterior se mostraría con las feas secuencias `\t` en lugar de la tabulación. Dentro de las cadenas entre comillas, solo el apóstrofe de escape (`\'`) y la propia barra invertida de escape (`\\`) se reconocen como caracteres de escape.

Comandos de varias líneas

Hay ocasiones en las que se necesita presentar una gran cantidad de texto de PHP, y usar varias declaraciones `echo` (o `print`) sería laborioso y engorroso. Para resolver este inconveniente PHP ofrece dos facilidades. La primera es poner varias líneas entre comillas, como en el Ejemplo 3-6. También se pueden asignar variables, como en el Ejemplo 3-7.

Ejemplo 3-6. Declaración echo de una cadena con varias líneas

```
<?php
  $author = "Steve Ballmer";

  echo "Developers, developers, developers, developers, developers,
  developers, developers, developers, developers!

  - $author.";
?>
```

Ejemplo 3-7. Asignación de una cadena con varias líneas

```
<?php
  $author = "Bill Gates";
  $text = "Measuring programming progress by lines of code is like
  Measuring aircraft building progress by weight.

  - $author.";
?>
```

PHP también puede tratar una secuencia de varias líneas mediante el operador <<<, al que normalmente nos referimos como *here-document* o *heredoc*, como una forma de especificar un literal de cadena, que preserva los saltos de línea y otros espacios en blanco (incluida la sangría) en el texto. Su uso se puede ver en el Ejemplo 3-8.

Ejemplo 3-8. Declaración echo alternativa de varias líneas

```
<?php
  $author = "Brian W. Kernighan";

  echo <<<_END
  Debugging is twice as hard as writing the code in the first place.
  Therefore, if you write the code as cleverly as possible, you are,
  by definition, not smart enough to debug it.

  - $author.
_END;
?>
```

Este código le dice a PHP que presente todo lo que hay entre las dos etiquetas _END como si fuera una cadena con comillas dobles (pero las comillas en un heredoc no necesitan caracteres de escape). Esto significa que es posible, por ejemplo, que un desarrollador escriba secciones enteras de HTML directamente en código PHP y luego reemplace partes dinámicas específicas con variables PHP.

Es importante recordar que el cierre _END; *debe* aparecer justo al comienzo de una nueva línea y debe ser *lo único* que haya en esa línea, ni siquiera se permite un comentario (ni siquiera un solo espacio). Una vez que hayamos cerrado un bloque de varias líneas, podemos usar el mismo nombre de etiqueta otra vez.

Recordemos: uso de <<<_END..._END. En el constructor heredoc no tienes que añadir caracteres \n para enviar un salto de línea, solo hay que pulsar Return e iniciar una nueva línea. Además, a diferencia de la delimitación de una cadena por comillas dobles o simples, se pueden usar las comillas simples y dobles que se quiera dentro de un heredoc, sin tener que anteponer la barra invertida (\).

El Ejemplo 3-9 muestra cómo usar la misma sintaxis para asignar varias líneas a una variable.

Ejemplo 3-9. Asignación a una variable de una cadena de varias líneas

```
<?php
  $author = "Scott Adams";

  $out = <<<_END
  Normal people believe that if it ain't broke, don't fix it.
  Engineers believe that if it ain't broke, it doesn't have enough
```

```
  features yet.

  - $author.
_END;
echo $out;
?>
```

El valor de la variable `$out` se rellenará entonces con el contenido entre las dos etiquetas. Si estuviéramos añadiendo en lugar de asignando, también podríamos haber usado `.=` en lugar de `=` para añadir la cadena a `$out`.

Hay que tener cuidado de no colocar un punto y coma directamente después de la primera aparición de `_END`, ya que eso terminaría el bloque de varias líneas antes de que comenzara y daría lugar a un mensaje de error de análisis. El único lugar para escribir el punto y coma es después de la etiqueta de terminación `_END`, aunque se puede usar el punto y coma dentro del bloque como un carácter de texto normal.

Por cierto, la etiqueta `_END` es simplemente una que elegí para estos ejemplos porque es poco probable que se utilice en cualquier otro lugar en código PHP y, por lo tanto, es única. Puedes usar cualquier etiqueta que te guste, como `_SECTION1` o `_OUTPUT`, etc. Además, para ayudar a diferenciar etiquetas como esta de variables o funciones, la práctica general es anteponer un guion bajo, pero no tenemos que usarlo si decidimos no hacerlo.

La colocación del texto en varias líneas es solo una conveniencia para hacer el código PHP más fácil de leer, porque una vez que se muestra en una página web, las reglas de formato HTML toman el control y el espacio en blanco se suprime (pero `$author` en nuestro ejemplo se seguirá sustituyendo por el valor de la variable).

Así, por ejemplo, si cargamos estos ejemplos de salida de varias líneas en un archivo, *no* se mostrarán varias líneas, ya que todos los navegadores tratan los saltos de línea como si fueran espacios. Sin embargo, si utilizamos la característica View Source del navegador, veremos que los saltos de línea está correctamente situados y que PHP mantiene los saltos de línea.

Tipificación de variables

PHP es un lenguaje débilmente tipado. Esto significa que las variables no se tienen que declarar antes de utilizarlas, y que PHP siempre convierte variables al tipo requerido por su contexto cuando se accede a ellas.

Por ejemplo, podemos crear un número de varios dígitos y extraer el enésimo dígito del mismo simplemente suponiendo que es una cadena. En el Ejemplo 3-10, los números `12345` y `67890` se multiplican, y se obtiene el resultado `838102050`, que se coloca en la variable `$number`.

Ejemplo 3-10. Conversión automática de un número en una cadena

```
<?php
  $number = 12345 * 67890;
  echo substr($number, 3, 1);
?>
```

En el momento de la asignación, `$number` es una variable numérica. Pero en la segunda línea, se hace una llamada a la función `substr` de PHP, que pide que se devuelva un carácter de `$number`, comenzando en la cuarta posición (recordemos que los desplazamientos en PHP empiezan en cero). Para hacer esto, PHP convierte `$number` en una cadena de nueve caracteres, de modo que `substr` puede acceder a ella y devolver el carácter, que en este caso es `1`.

Lo mismo se aplica a la conversión de una cadena en un número, etc. En el Ejemplo 3-11, el valor de la variable `$pi` es una cadena, que luego se convierte automáticamente en un número de punto flotante en la tercera línea, que utiliza la ecuación para calcular el área de un círculo, que da como resultado el valor `78.5398175`.

Ejemplo 3-11. Conversión automática de una cadena en un número

```
<?php
  $pi      = "3.1415927";
  $radius  = 5;
  echo $pi * ($radius * $radius);
?>
```

En la práctica, lo que todo esto significa es que no hay que preocuparse demasiado por los tipos de variables. Simplemente asignamos valores que tengan sentido para nosotros, y PHP los convertirá si es necesario. Después, si queremos extraer valores, solo tenemos que pedirlos, por ejemplo, con una declaración `echo`.

Constantes

Las *constantes* son similares a las variables, contienen información a la que se accede después, excepto que son lo que parecen: constantes. En otras palabras, una vez que hayas definido una, su valor se fija para el resto del programa y no se puede cambiar.

Un ejemplo de uso de una constante es el de la ubicación de la carpeta principal del servidor (la carpeta con los archivos principales de tu sitio web). Definirías una constante como esta:

```
define("ROOT_LOCATION", "/usr/local/www/");
```

Luego, para leer el contenido de la variable, solo tienes que referirte a ella como una variable normal (pero la constante no está precedida por el símbolo del dólar):

```
$directory = ROOT_LOCATION;
```

Después de esto, siempre que necesitemos ejecutar código PHP en un servidor diferente con una configuración de carpetas diferente, solo tenemos que cambiar una sola línea de código.

Las dos cosas principales que debes recordar sobre las constantes son que no deben ir precedidas de $ (a diferencia de las variables normales) y que solo podemos definirlas con la función `define`.

Generalmente se considera una buena práctica usar solo letras mayúsculas para nombres de variables constantes, especialmente si otras personas también leen tu código.

Constantes predefinidas

PHP viene preparado con docenas de constantes predefinidas que por lo general no utilizarás al principio. Sin embargo, hay unas pocas, conocidas como *constantes mágicas*, que te resultarán útiles. Los nombres de las constantes mágicas siempre tienen dos guiones bajos en el campo al principio y dos al final, para que no intentes accidentalmente nombrar uno de tus constantes propias con un nombre que ya está ocupado. Se detallan en la Tabla 3-5. Los conceptos a los que se hace referencia en el cuadro se introducirán en futuros capítulos.

Tabla 3-5. Constantes mágicas de PHP

Constante mágica	Descripción
`__LINE__`	Número de la línea actual del archivo.
`__FILE__`	Ruta completa y nombre del archivo. Si se usa dentro de `include`, devuelve el nombre del archivo incluido. Algunos sistemas operativos permiten usar alias para directorios a los que se les llama enlaces simbólicos. En `__FILE__` estos siempre se cambian a los directorios reales.
`__DIR__`	Directorio del archivo. (Añadido en PHP 5.3.0). Si se usa dentro de `include`, se devuelve el directorio del archivo incluido. Es equivalente a `dirname(__FILE__)`. Este nombre de directorio no tiene la barra inclinada a menos que sea el directorio raíz.
`__FUNCTION__`	Nombre de la función. (Añadido en PHP 4.3.0). A partir de PHP 5, devuelve el nombre de la función tal como se ha declarado (distingue entre mayúsculas y minúsculas). En PHP 4, siempre va en minúsculas.
`__CLASS__`	Nombre de la clase. (Añadido en PHP 4.3.0). A partir de PHP 5, devuelve el nombre de la clase tal como se ha declarado (distingue entre mayúsculas y minúsculas). En PHP 4, siempre va en minúsculas.
`__METHOD__`	Nombre del método de la clase. (Añadido en PHP 5.0.0). El nombre del método se devuelve tal como se ha declarado (distingue entre mayúsculas y minúsculas).
`__NAMESPACE__`	Nombre del actual espacio de nombres. (Añadido en PHP 5.3.0.) Esta constante se define en la compilación (distingue entre mayúsculas y minúsculas).

Estas variables encuentran un uso práctico en la depuración, cuando se necesita insertar una línea de código para ver si el flujo del programa llega hasta ella:

```
echo "This is line " . __LINE__ . " of file " . __FILE__;
```

Esto imprime la línea del programa en curso del archivo de trabajo (incluida la ruta) en la ventana de diálogo del navegador web.

Diferencia entre los comandos echo y print

Hasta ahora hemos visto el comando `echo` que hemos utilizado de diferentes maneras para enviar texto desde el servidor al navegador. En algunos casos, la salida ha sido un literal de cadena. En otros, las cadenas se han concatenado o se han evaluado las variables, antes de utilizar el comando. También he explicado las salidas con varias líneas.

Pero hay una alternativa a `echo` que podemos usar: `print`. Los dos comandos son bastante similares, pero `print` es un constructor parecido a una función que admite un solo parámetro y tiene un valor de retorno (que siempre es `1`), mientras que `echo` es puramente un constructor del lenguaje PHP. Dado que ambos comandos son constructores, ninguno requiere paréntesis.

En general, el comando `echo` normalmente será un poco más rápido que `print`, porque no establece un valor de retorno. Por otro lado, debido a que `echo` no se implementa como una función, no se puede usar como parte de una expresión más compleja, mientras que `print` sí.

He aquí un ejemplo para determinar si el valor de una variable es `TRUE` o `FALSE` usando `print`, algo que no se podría realizar de la misma manera con `echo`, ya que mostraría un mensaje de `Parse error` (error de análisis).

```
$b ? print "TRUE" : print "FALSE";
```

El signo de interrogación es simplemente una forma de preguntar si la variable `$b` es `TRUE` o `FALSE`. El comando que está a la izquierda de los dos puntos, se ejecuta si `$b` es `TRUE`, mientras que el comando a la derecha de los dos puntos se ejecuta si `$b` es `FALSE`.

Generalmente, sin embargo, los ejemplos en este libro usan `echo`, y recomiendo que lo hagas así hasta que llegues a tal punto en tu desarrollo PHP que descubras la necesidad de utilizar `print`.

Funciones

Las *funciones* contienen las secciones de código que realizan una tarea en concreto. Por ejemplo, tal vez a menudo necesites buscar una fecha y presentarla en un formato determinado. Este podría ser un buen ejemplo para convertirla en una función. El código correspondiente puede tener solo tres líneas, pero si tienes que pegarlo en tu programa una docena de veces, estás haciendo que este sea innecesariamente grande y complejo si no utilizas una función. Y si decides cambiar el formato de fecha más tarde, ponerlo en una función significa tener que cambiarlo en un solo lugar.

Crear una función con un código que se utiliza a menudo no solo acorta el programa y lo hace más legible, sino que también añade funcionalidad extra (juego de palabras), ya

que se pueden pasar parámetros a las funciones para que operen de manera diferente. También pueden devolver valores al código desde el que se las llama.

Para crear una función, hay que declararla como se muestra en el Ejemplo 3-12.

Ejemplo 3-12. Declaración de una función sencilla

```
<?php
  function longdate($timestamp)
  {
    return date("l F jS Y", $timestamp);
  }
?>
```

Esta función devuelve una fecha con el formato *Sunday May 2nd 2021 (*domingo 2 de mayo de 2021). Se puede incluir cualquier número de parámetros entre los paréntesis, pero en este caso hemos optado por definir solo uno. Las llaves encierran el código que se ejecuta cuando más tarde se llama a la función. Observa que la primera letra dentro de la llamada a la función `date` en este ejemplo es una letra minúscula ele "L", que no debe confundirse con el número 1.

Para presentar la fecha de hoy mediante esta función, escribe la siguiente llamada en tu código:

```
echo longdate(time());
```

Si necesitas imprimir la fecha de hace 17 días, solo tienes que realizar la siguiente llamada:

```
echo longdate(time() - 17 * 24 * 60 * 60);
```

que pasa a `longdate` la hora actual menos el número de segundos desde hace 17 días (17 días × 24 horas × 60 minutos × 60 segundos).

Las funciones también pueden aceptar varios parámetros y devolver varios resultados mediante técnicas que presentaré en los siguientes capítulos.

Ámbito de aplicación de variables

Si tienes un programa muy largo, es muy posible que empiecen a escasear los buenos nombres de variables, pero con PHP puedes decidir el *scope* (ámbito) de una variable. En otras palabras, puedes, por ejemplo, decirle que deseas que la variable `$temp` solo se utilice dentro de una función en particular y olvidarte de que la variable se ha utilizado cuando la función entrega el resultado. De hecho, este es el ámbito por defecto para las variables PHP.

Opcionalmente, puedes informar a PHP que una variable es de ámbito global y por lo tanto se puede acceder a ella desde cualquier otra parte del programa.

Variables locales

Las *variables locales* son variables que se crean dentro de una función y a las que solo se puede acceder mediante esa función. Generalmente son variables temporales que se

utilizan para almacenar resultados parcialmente procesados antes de que la función proporcione un resultado.

La lista de argumentos de una función es un conjunto de variables locales. En la sección anterior, definimos una función que aceptaba un parámetro llamado `$timestamp`. Este solo tiene sentido en el cuerpo de la función; no se puede obtener o establecer su valor fuera de la función.

Para ver otro ejemplo de una variable local, echa otro vistazo a la función `longdate`, que se ha modificado ligeramente en el Ejemplo 3-13.

Ejemplo 3-13. Una versión ampliada de la función longdate

```
<?php
  function longdate($timestamp)
  {
    $temp = date("l F jS Y", $timestamp);
    return "The date is $temp";
  }
?>
```

Aquí hemos asignado el valor devuelto por la función `date` (fecha) a la variable temporal `$temp`, que luego se inserta en la cadena devuelta por la función. Tan pronto como la función la devuelve, la variable `$temp` y su contenido desaparecen, como si nunca se hubieran utilizado.

Ahora, para ver los efectos del ámbito de la variable, veamos algún código similar en el Ejemplo 3-14. Aquí se ha creado `$temp` *antes* de que llamemos a la función `longdate`.

Ejemplo 3-14. Este intento de acceder a $temp in la función longdate no tendrádaxito

```
<?php
  $temp = "The date is ";
  echo longdate(time());

  function longdate($timestamp)
  {
    return $temp . date("l F jS Y", $timestamp);
  }
?>
```

Sin embargo, debido a que `$temp` no se ha creado dentro de la función `longdate` ni se le ha pasado como parámetro, `longdate` no puede acceder a ella. Por lo tanto, este fragmento de código solo produce la fecha, no el texto precedente. De hecho, en función de cómo esté configurado PHP, puede mostrar primero el mensaje de error `Notice: Undefined variable: temp` (Aviso: Variable indefinida: temp), algo que no quieres que tus usuarios vean.

La razón de esto es que, por defecto, las variables creadas dentro de una función son locales de esa función, y a las variables creadas fuera de cualquier función solo se puede acceder por un código que no sea una función.

Algunas maneras de hacer que funcione el Ejemplo 3-14 se muestran en los ejemplos 3-15 y 3-16.

Ejemplo 3-15. Con la reescritura del código para referirse a $temp dentro de su ámbito local se resuelve el problema

```
<?php
  $temp = "The date is ";
  echo $temp . longdate(time());

  function longdate($timestamp)
  {
    return date("l F jS Y", $timestamp);
  }
?>
```

En el Ejemplo 3-15 se saca la referencia a `$temp` de la función. La referencia aparece en el mismo ámbito en el que se definió la variable.

Ejemplo 3-16. Una solución alternativa: pasar $temp como argumento

```
<?php
  $temp = "The date is ";
  echo longdate($temp, time());

  function longdate($text, $timestamp)
  {
    return $text . date("l F jS Y", $timestamp);
  }
?>
```

La solución del Ejemplo 3-16 pasa `$temp` a la función `longdate` como argumento extra. `longdate` lo lee en una variable temporal que crea llamada `$text` y produce el resultado deseado.

Olvidar el ámbito de una variable es un error de programación muy corriente, así que recordar cómo funciona el ámbito de la variable te ayudará a depurar algunos problemas bastante complicados. Baste decir que, a menos que se haya declarado una variable de otro modo, su alcance se limita al ámbito local: ya se trate de la actual función o de código fuera de cualquier función, dependiendo de si se creó por primera vez o si se accedió a ella, desde el interior de una función o desde fuera de la misma.

Variables globales

Hay casos en los que se necesita que una variable tenga un ámbito *global*, porque se desea que cualquier parte del código pueda acceder a ella. Además, algunos datos pueden ser voluminosos y complejos, y no quieres seguir pasándolos como argumentos a las funciones. Para acceder a variables de alcance global, añade la palabra clave `global`. Supongamos que tienes una manera de registrar a tus usuarios en tu sitio web y quieres que tu código

sepa si está interactuando con un usuario conectado o con un invitado. Una manera de hacerlo es usar la palabra clave `global` delante de una variable como `$is_logged_in`:

```
global $is_logged_in;
```

Ahora tu función de inicio de sesión simplemente tiene que establecer esa variable a `1` en un inicio de sesión con éxito o `0` en caso de fallo. Debido a que el alcance de la variable se define como global, cada línea de código de tu programa puede acceder a ella.

Sin embargo, debes utilizar con precaución las variables a las que se ha dado acceso global. Te recomiendo que las crees solo cuando no puedas encontrar otra manera de lograr el resultado que deseas. En general, los programas que están divididos en partes pequeñas y tienen datos segregados tienen menos errores y son más fáciles de mantener. Si tienes un programa de mil líneas (y algún día lo tendrás) en el que descubres que una variable global tiene un valor equivocado en algún momento, ¿cuánto tiempo necesitarás para encontrar el código que la configuró incorrectamente?

Además, si tienes demasiadas variables de alcance global, corres el riesgo de usar uno de esos nombres de nuevo localmente, o al menos pensar que lo has usado localmente, cuando en realidad ya se ha declarado como global. De estas situaciones pueden surgir todo tipo de errores extraños.

A veces adopto la convención de escribir con mayúsculas los nombres de variables que requieren acceso global (así como se recomienda que las constantes deben estar en mayúsculas) para poder saber de un vistazo el alcance de una variable.

Variables estáticas

En el apartado "Variables locales" en la página 55, mencioné que el valor de una variable local se borra una vez que la función ha entregado un resultado. Si una función se ejecuta muchas veces, comienza con una nueva copia de la variable y la configuración anterior no tiene ningún efecto.

Este es un caso interesante. ¿Qué pasa si tienes una variable local dentro de una función a la que no quieres que tengan acceso otras partes de tu código, pero también te gustaría mantener su valor para la próxima vez que se llame a la función? ¿Por qué? Tal vez porque quieres que un contador rastree cuántas veces se llama a una función. La solución es declarar una variable *estática*, como se muestra en el Ejemplo 3-17.

Ejemplo 3-17. Función que utiliza una variable estática

```
<?php
  function test()
  {
    static $count = 0;
    echo $count;
    $count++;
  }
?>
```

Aquí, la primera línea de la función `test` crea una variable estática llamada `$count` y la inicializa a un valor de `0`. La siguiente línea presenta el valor de la variable; la línea final lo incrementa.

La próxima vez que se llame a la función, como ya se ha declarado `$count`, se salta la primera línea de la función. A continuación, muestra el valor incrementado previamente de `$count` antes de que la variable vuelva a incrementarse.

Si planeas utilizar variables estáticas, debes tener en cuenta que no les puedes asignar el resultado de una expresión cuando las defines. Solo se pueden inicializar con valores predeterminados (ver el Ejemplo 3-18).

Ejemplo 3-18. Declaraciones permitidas y no permitidas de variables estáticas

```
<?php
  static $int = 0;          // Allowed
  static $int = 1+2;        // Disallowed (will produce a parse
error)
  static $int = sqrt(144);  // Disallowed
?>
```

Variables superglobales

A partir de PHP 4.1.0, están disponibles varias variables predefinidas. A estas se las conoce como *variables superglobales*, lo que significa que las proporciona el entorno PHP, pero son globales dentro del programa, accesibles absolutamente desde todas partes.

Estas superglobales contienen mucha información útil sobre el programa en ejecución y su entorno (ver Tabla 3-6). Están estructuradas como matrices asociativas, un tema que se estudia en el Capítulo 6.

Tabla 3-6. Variables superglobales de PHP

Nombre	Contenidos
`$GLOBALS`	Todas las variables que están definidas en ese momento en el ámbito global del script. Los nombres de las variables son las claves de la matriz.
`$_SERVER`	Información como encabezados, rutas y ubicaciones de scripts. Las entradas en esta matriz las crea el servidor web, y no hay garantías de que todos los servidores web proporcionen alguna o todas estas entradas.
`$_GET`	Variables pasadas al script en curso mediante el método HTTP GET
`$_POST`	Variables pasadas al script en curso mediante el método HTTP POST
`$_FILES`	Elementos cargados en el script en curso mediante el método HTTP POST
`$_COOKIE`	Variables pasadas al script en curso a través de cookies HTTP
`$_SESSION`	Variables de sesión disponibles para el script en curso
`$_REQUEST`	Contenido de la información pasada desde el navegador; por defecto, `$_GET`, `$_POST`, y `$_COOKIE`
`$_ENV`	Variables pasadas al script en curso mediante el método del entorno

Todas las superglobales (excepto `$GLOBALS`) se nombran con un único guion bajo inicial y se escriben solo con letras mayúsculas; por lo tanto, debes evitar nombrar tus propias variables de esta manera para evitar posibles confusiones.

Para ilustrar cómo las debes usar, veamos un ejemplo corriente. Entre las muchas perlas de información que suministran las variables superglobales se encuentra el URL de la página que remitió al usuario a la página web en la que se encuentra. Se puede acceder a esta información de la página de referencia de la siguiente manera:

```
$came_from = $_SERVER['HTTP_REFERER'];
```

Es así de simple. Oh, y si el usuario llegó directamente a tu página web escribiendo tu URL en un navegador, `$came_from` tendrá una cadena vacía.

Superglobales y seguridad

Hay que tener cuidado antes de empezar a usar variables superglobales, porque los hackers las utilizan a menudo tratando de encontrar vulnerabilidades para entrar en los sitios web. Lo que hacen es cargar `$_POST`, `$_GET` u otras superglobales con código malicioso, tales como comandos Unix o MySQL que pueden dañar o mostrar datos sensibles si ingenuamente accedes a ellos.

Por lo tanto, siempre debes desinfectar las superglobales antes de usarlas. Una forma de hacerlo es a través de la función `htmlentities` de PHP. Convierte todos los caracteres en entidades HTML. Por ejemplo, los caracteres menor que y mayor que (< y >) se transforman en las cadenas `<` y `>` de modo que son inofensivos, como lo son todas las comillas y barras invertidas, etc.

Por lo tanto, una forma mucho más conveniente de acceder a `$_SERVER` (y otras superglobales) es:

```
$came_from = htmlentities($_SERVER['HTTP_REFERER']);
```

El uso de la función `htmlentities` para la desinfección es una práctica importante en cualquier circunstancia en la que los datos del usuario o de terceros se procesan para su salida, no solo con superglobales.

Este capítulo te ha proporcionado una sólida introducción al uso de PHP. En el Capítulo 4, comenzarás a usar lo que has aprendido para crear expresiones y controlar el flujo del programa; en otras palabras, hacer programación real.

Pero antes de continuar, te recomiendo que te evalúes con algunas (ni no todas) de las siguientes preguntas, para tener la seguridad de que has digerido por completo el contenido de este capítulo.

Preguntas

1. ¿Qué etiqueta se usa para llamar a PHP para que comience a interpretar el código del programa? ¿Y cuál es la versión corta de la etiqueta?
2. ¿Cuáles son los dos tipos de etiquetas de comentarios?
3. ¿Qué carácter debe colocarse al final de cada declaración PHP?
4. ¿Qué símbolo se antepone a todas las variables de PHP?
5. ¿Qué puede almacenar una variable?
6. ¿Cuál es la diferencia entre `$variable = 1` y `$variable == 1`?
7. ¿Por qué se supone que se permite un guion bajo en nombres de variables (`$current_user`), mientras que los guiones no (`$current-user`)?
8. ¿Los nombres de las variables distinguen entre mayúsculas y minúsculas?
9. ¿Se pueden utilizar espacios en nombres de variables?
10. ¿Cómo se convierte un tipo de variable a otro (digamos, una cadena a un número)?
11. ¿Cuál es la diferencia entre `++$j` y `$j++`?
12. ¿Son los operadores `&&` y `and` intercambiables?
13. ¿Cómo se puede crear un `echo` o una asignación de varias líneas?
14. ¿Puedes redefinir una constante?
15. ¿Cómo se puede escapar de una comilla?
16. ¿Cuál es la diferencia entre los comandos `echo` y `print`?
17. ¿Cuál es el propósito de las funciones?
18. ¿Cómo puedes hacer que una variable sea accesible desde cualquier parte de un programa PHP?
19. Si generas datos dentro de una función, ¿cuáles son un par de formas de transmitirla al resto del programa?
20. ¿Cuál es el resultado de combinar una cadena con un número?

Consulta "Respuestas del Capítulo 3" en la página 706 en el Apéndice A para comprobar las respuestas a estas preguntas.

CAPÍTULO 4

Expresiones y control de flujo en PHP

En el capítulo anterior se introdujeron de pasada varios temas que este capítulo trata de manera más completa, como son la toma de decisiones (ramificación) y la creación de expresiones complejas. En el capítulo anterior, quise centrarme en la sintaxis y operaciones más básicas en PHP, pero no pude evitar tocar temas más avanzados. Ahora puedo completar los antecedentes que necesitas para usar estas potentes características de PHP correctamente.

En este capítulo, adquirirás unos completos conocimientos de cómo funciona la programación PHP en la práctica y de cómo controlar el flujo del programa.

Expresiones

Comencemos con la parte más fundamental de cualquier lenguaje de programación: las *expresiones*.

Una expresión es una combinación de valores, variables, operadores y funciones que presenta como resultado un valor. Resultará familiar para cualquiera que haya estudiado álgebra en la enseñanza secundaria. Aquí hay un ejemplo:

```
y = 3 (|2x| + 4)
```

Que en PHP sería:

```
$y = 3 * (abs(2 * $x) + 4);
```

El valor generado (`y` en la declaración matemática, o `$y` en el PHP) puede ser un número, una cadena o un valor *booleano* (bautizado en honor a George Boole, un matemático y filósofo inglés del siglo XIX). A estas alturas, deberías estar familiarizado con los dos primeros tipos de valores, y ahora explicaré el tercero.

¿TRUE o FALSE?

Un valor booleano básico puede ser `TRUE` (VERDADERO) o `FALSE` (FALSO). Por ejemplo, la expresión `20 > 9` (20 es mayor que 9) es `TRUE`, y la expresión `5 == 6` (5 es igual a 6) es `FALSE`. (Puedes combinar estas operaciones con otros operadores booleanos clásicos como `AND`, `OR` y `XOR`, que se tratan más adelante en este capítulo.

Como puedes ver, utilizo letras mayúsculas para los nombres `TRUE` y `FALSE`. Esto se debe a que son constantes predefinidas en PHP. Tú puedes utilizar las versiones en minúsculas si lo prefieres, ya que también están predefinidas. De hecho, las versiones en minúsculas son más estables, porque PHP no te permite redefinirlas. Las mayúsculas se pueden redefinir, que es algo que debes tener en cuenta si importas código de terceros.

PHP no imprime las constantes predefinidas aunque le pidas que lo haga, como en el Ejemplo 4-1. Para cada línea, el ejemplo imprime una letra seguida de dos puntos y una constante predefinida. PHP asigna arbitrariamente el valor numérico de `1` a `TRUE`, de modo que cuando se ejecuta el ejemplo, el 1 aparece después de `a:`. Incluso todavía hay más misterio, porque `b:` evalúa a `FALSE` y no muestra ningún valor. En PHP la constante `FALSE` se define como `NULL`, otra constante predefinida que no significa nada.

Ejemplo 4-1. Salida de los valores TRUE y FALSE

```
<?php // test2.php
  echo "a: [" . TRUE  . "]<br>";
  echo "b: [" . FALSE . "]<br>";
?>
```

Las etiquetas `<br>` están ahí para crear saltos de línea y así separar el resultado en dos líneas en HTML. Aquí está el resultado:

```
a: [1]
b: []
```

Volvamos a las expresiones booleanas; el Ejemplo 4-2 muestra algunas expresiones sencillas: las dos que mencioné anteriormente, y un par de ellas más.

Ejemplo 4-2. Cuatro expresiones booleanas sencillas

```
<?php
  echo "a: [" . (20 > 9)  . "]<br>";
  echo "b: [" . (5 == 6)  . "]<br>";
  echo "c: [" . (1 == 0)  . "]<br>";
  echo "d: [" . (1 == 1)  . "]<br>";
?>
```

El resultado de este código es:

```
a: [1]
b: []
c: []
d: [1]
```

Por cierto, en algunos lenguajes `FALSE` puede definirse como `0` o incluso `-1`, por lo que vale la pena comprobar su definición en cada uno de los lenguajes que utilices. Afortunadamente, las expresiones booleanas están normalmente ocultas en otros códigos, así que, por lo general, no tienes que preocuparte por cómo se vean internamente `TRUE` y `FALSE`. De hecho, incluso esos nombres raramente aparecen en el código.

Literales y variables

Estos son los elementos básicos empleados en programación, y los componentes de las expresiones. Un *literal* es simplemente algo que se evalúa a sí mismo, como el número `73` o la cadena `"Hello"`. Una variable, que como ya hemos visto tiene un nombre que comienza con el símbolo del dólar, evalúa el valor que se le ha asignado. La expresión más sencilla es la que está formada por un único literal o una variable, porque ambos devuelven un valor.

El Ejemplo 4-3 muestra tres literales y dos variables, todas ellos devuelven valores, si bien de diferentes tipos.

Ejemplo 4-3. Literales y variables

```
<?php
  $myname = "Brian";
  $myage = 37;

  echo "a: " . 73        . "<br>"; // Numeric literal
  echo "b: " . "Hello"   . "<br>"; // String literal
  echo "c: " . FALSE     . "<br>"; // Constant literal
  echo "d: " . $myname   . "<br>"; // String variable
  echo "e: " . $myage    . "<br>"; // Numeric variable
?>
```

Y, como es de esperar, todos ellos proporcionan un valor como respuesta con la excepción de `c:`, que se evalúa a `FALSE`, y no devuelve nada en la siguiente salida:

```
a: 73
b: Hello
c:
d: Brian
e: 37
```

Junto con los operadores, es posible crear expresiones más complejas que evalúen resultados útiles.

Los programadores combinan expresiones con otros constructores del lenguaje, como los operadores de asignación que vimos anteriormente, para formar *declaraciones*. El Ejemplo 4-4 muestra dos declaraciones. La primera asigna el resultado de la expresión `366 - $day_number` a la variable `$days_to_new_year`, y la segunda presenta un mensaje amistoso solo si la expresión `$days_to_new_year < 30` evalúa a `TRUE`.

Ejemplo 4-4. Una expresión y una declaración

```
<?php
  $days_to_new_year = 366 - $day_number; // Expression

  if ($days_to_new_year < 30)
  {
    echo "Not long now till new year"; // Statement
  }
?>
```

Operadores

PHP ofrece una gran cantidad de operadores muy potentes de diferentes tipos (aritméticos, de cadenas de caracteres, lógicos, de asignación, de comparación, etc. (ver Tabla 4-1).

Tabla 4-1. Tipos de operadores de PHP

Operador	Descripción	Ejemplo
Aritmético	Matemáticas básicas	`$a + $b`
Matriz	Unión de matrices	`$a + $b`
Asignación	Asigna valores	`$a = $b + 23`
Bitwise	Manipula bits dentro de bytes	`12 ^ 9`
Comparación	Compara dos valores	`$a < $b`
Ejecución	Ejecuta contenido de las comillas	`` `ls -al` ``
Incremento/decremento	Suma o resta 1	`$a++`
Lógico	Booleano	`$a and $b`
Cadena	Concatenación	`$a . $b`

Cada operador admite un número diferente de operandos:

- Operadores *unarios*, como el incremento (`$a++`) o la negación (`!$a`), utilizan un solo operando.
- Operadores *binarios*, que representan la mayor parte de los operadores PHP (incluidas sumas, restas, multiplicaciones y divisiones), utilizan dos operandos.
- El único operador *ternario*, que toma la forma `expr ? x : y`, requiere tres operandos. Es una breve declaración `if`, de una sola línea, que devuelve `x` si `expr` es `TRUE` e `y` si `expr` es `FALSE`.

Prioridades de los operadores

Si todos los operadores tuvieran la misma prioridad, se procesarían en el orden en que se encuentran. De hecho, muchos operadores tienen la misma prioridad. Echa un vistazo al Ejemplo 4-5.

Ejemplo 4-5. Tres expresiones equivalentes

```
1 + 2 + 3 - 4 + 5
2 - 4 + 5 + 3 + 1
5 + 2 - 4 + 1 + 3
```

Aquí verás que aunque los números (y sus operadores precedentes) han cambiado de lugar, el resultado de cada expresión es el valor `7`, porque los operadores más y menos tienen la misma prioridad. Podemos intentar lo mismo con la multiplicación y la división (ver Ejemplo 4-6).

Ejemplo 4-6. Tres expresiones que son también equivalentes

```
1 * 2 * 3 / 4 * 5
2 / 4 * 5 * 3 * 1
5 * 2 / 4 * 1 * 3
```

Aquí el valor resultante es siempre `7.5`. Pero las cosas cambian cuando mezclamos operadores con *diferentes* prioridades en una expresión, como las del Ejemplo 4-7.

Ejemplo 4-7. Tres expresiones que utilizan operadores con diferentes prioridades

```
1 + 2 * 3 - 4 * 5
2 - 4 * 5 * 3 + 1
5 + 2 - 4 + 1 * 3
```

Si los operadores no tuvieran prioridades, los valores de estas expresiones serían `25`, `-29` y `12`, respectivamente. Pero debido a que la multiplicación y la división tienen prioridad sobre la suma y la resta, las expresiones se evalúan como si hubiera paréntesis que contienen estas partes de las expresiones, como en el caso de la notación matemática (ver el Ejemplo 4-8).

Ejemplo 4-8. Tres expresiones que muestran paréntesis implícitos

```
1 + (2 * 3) - (4 * 5)
2 - (4 * 5 * 3) + 1
5 + 2 - 4 + (1 * 3)
```

En primer lugar PHP evalúa las subexpresiones entre paréntesis para mostrar los resultados parciales que aparecen el Ejemplo 4-9.

Ejemplo 4-9. Después de evaluar las subexpresiones entre paréntesis

```
1 + (6) - (20)
2 - (60) + 1
5 + 2 - 4 + (3)
```

Los resultados finales de estas expresiones son `-13`, `-57` y `6`, respectivamente (bastante diferentes de los resultados de `25`, `-29` y `12` que habríamos obtenido si no existiera la prioridad de operadores).

Por supuesto, puedes anular la prioridad por defecto insertando tus propios paréntesis y forzar el orden que quieras (ver Ejemplo 4-10).

Ejemplo 4-10. Cómo forzar la evaluación de izquierda a derecha

```
((1 + 2) * 3 - 4) * 5
(2 - 4) * 5 * 3 + 1
(5 + 2 - 4 + 1) * 3
```

Con los paréntesis utilizados correctamente, ahora vemos los valores `25`, `-29` y `12`, respectivamente.

La Tabla 4-2 lista los operadores de PHP en orden de prioridad del más alto al más bajo.

Tabla 4-2. Prioridad de los operadores de PHP (de mayor a menor)

Operador(es)	Tipo
( )	Paréntesis
++ --	Incremento/decremento
!	Lógico
* / %	Aritmético
+ - .	Aritmético y de cadenas
<< >>	Bit a bit
< <= > >= <>	Comparación
== != === !==	Comparación
&	Bit a bit (y referencias)
^	Bit a bit
|	Bit a bit
&&	Lógico
||	Lógico
? :	Ternario
= += -= *= /= .= %= &= != ^= <<= >>=	Asignación
and	Lógico
xor	Lógico
or	Lógico

El orden en esta tabla no es arbitrario, sino que está cuidadosamente diseñado para que las prioridades más habituales e intuitivas sean las que se pueden obtener sin paréntesis. Por ejemplo, puedes separar dos comparaciones con `and` o con `or` y obtener el resultado que cabría esperar según la prioridad.

Asociatividad

Hemos estado viendo cómo se procesan las expresiones de izquierda a derecha, excepto cuando la prioridad del operador entra en juego. Pero algunos operadores requieren un procesamiento de derecha a izquierda, y esta dirección de procesamiento se denomina *asociatividad* del operador. Para algunos operadores, no hay asociatividad.

La asociatividad (como se detalla en la Tabla 4-3) adquiere importancia en los casos en los que no se fuerza explícitamente la prioridad, por lo que es necesario estar al tanto de las acciones por defecto de los operadores.

Tabla 4-3. Asociatividad de los operadores

Operador	Descripción	Asociatividad
`< <= >= == != === !== <>`	Comparación	No
`!`	NO lógico	Derecha
`~`	NO a nivel de bit	Derecha
`++ --`	Incremento y decremento	Derecha
`(int)`	Convierte a un entero	Derecha
`(double) (float) (real)`	Convierte a un nº en punto flotante	Derecha
`(string)`	Convierte a una cadena	Derecha
`(array)`	Convierte a una matriz	Derecha
`(object)`	Convierte a un objeto	Derecha
`@`	Inhibe el reporte de errores	Derecha
`= += -= *= /=`	Asignación	Derecha
`.= %= &= |= ^= <<= >>=`	Asignación	Derecha
`+`	Adición y unario más	Izquierda
`-`	Sustracción y negación	Izquierda
`*`	Multiplicación	Izquierda
`/`	División	Izquierda
`%`	Módulo	Izquierda
`.`	Concatenación cadenas	Izquierda
`<< >> & ^ |`	A nivel de bit	Izquierda
`?:`	Ternario	Izquierda
`|| && and or xor`	Lógico	Izquierda
`,`	Separador	Izquierda

Por ejemplo, echemos un vistazo al operador de asignación en el Ejemplo 4-11, en el que tres variables tienen todas el valor 0.

Ejemplo 4-11. Declaración de asignación múltiple

```
<?php
  $level = $score = $time = 0;
?>
```

Esta asignación múltiple solo es posible si primero se evalúa la parte más a la derecha de la expresión y, a continuación, el procesamiento continúa en la dirección de derecha a izquierda.

Como recién llegado a PHP, debes evitar los riesgos potenciales de anidar siempre las subexpresiones entre paréntesis para forzar el orden de evaluación. Esto también ayudará a otros programadores, que pueden tener que mantener tu código, a entender lo que ocurre.

Operadores relacionales

Los operadores relacionales responden a preguntas como "¿Tiene esta variable un valor de cero?" y "¿Qué variable tiene mayor valor?". Estos operadores comprueban dos operandos y devuelven un resultado booleano de `TRUE` o `FALSE`. Existen tres tipos de operadores relacionales: de *igualdad*, de *comparación* y *lógicos*.

Igualdad

Como ya hemos visto varias veces en este capítulo, el operador de igualdad es `==` (dos signos de igualdad). Es importante no confundirlo con el operador de asignación `=` (signo de igualdad). En el Ejemplo 4-12, la primera declaración asigna un valor y la segunda prueba su igualdad.

Ejemplo 4-12. Asignación de un valor y comprobación de igualdad

```
<?php
  $month = "March";

  if ($month == "March") echo "It's springtime";
?>
```

Como puedes ver, al devolver `TRUE` o `FALSE`, el operador de igualdad permite probar las condiciones mediante, por ejemplo, una declaración `if`. Pero esa no es toda la historia, porque PHP es un lenguaje poco estructurado. Si los dos operandos de una expresión de igualdad son de diferentes tipos, PHP los convertirá a cualquiera de los dos tipos que mejor se adapte a sus necesidades. Se puede hacer uso de un operador de *identidad* que raramente se utiliza, que consiste en tres signos iguales seguidos, para comparar elementos sin realizar la conversión.

Por ejemplo, cualquier cadena compuesta totalmente por números se convertirá en números cuando se compara con un número. En el Ejemplo 4-13, `$a` y `$b` son dos cadenas diferentes y, por lo tanto, no esperamos que ninguna de las sentencias `if` produzca un resultado.

Ejemplo 4-13. Operadores de igualdad y de identidad

```
<?php
  $a = "1000";
  $b = "+1000";

  if ($a == $b) echo "1";
  if ($a === $b) echo "2";
?>
```

Sin embargo, si ejecutas el ejemplo, verás que da como resultado el número `1`, lo cual significa que la primera declaración `if` se ha evaluado como `TRUE`. Esto se debe a que ambas cadenas se han convertido primero a números, y `1000` es el mismo valor numérico que `+1000`. En contraste, la segunda declaración `if` usa el operador de

identidad, por lo que compara $a y $b como cadenas, ve que son diferentes, y por lo tanto no presenta ningún resultado.

Como en el caso en el que se fuerza la prioridad del operador, siempre que tengas alguna duda sobre cómo convertirá PHP los tipos de operandos, puedes utilizar el operador de identidad para desactivar la conversión.

De la misma manera que puedes utilizar el operador de igualdad para comprobar si los operandos son iguales, puedes comprobar que *no* son iguales si usas el operador de desigualdad. Echa un vistazo al Ejemplo 4-14, que es una reescritura del Ejemplo 4-13, en el que los operadores de igualdad y de identidad se han sustituido por sus opuestos.

Ejemplo 4-14. Desigualdad y operadores no idénticos

```
<?php
  $a = "1000";
  $b = "+1000";

  if ($a != $b)  echo "1";
  if ($a !== $b) echo "2";
?>
```

Y, como es de esperar, la primera sentencia `if` no da como resultado el número `1`, porque el código está preguntando si `$a` y `$b` *no* son iguales numéricamente.

En su lugar, este código da como resultado el número `2`, porque la segunda declaración `if` pregunta si `$a` y `$b` *no* son idénticos entre sí en su tipo de cadena real, y la respuesta es `TRUE`; no son lo mismo.

Operadores de comparación

Mediante el uso de operadores de comparación, se puede comprobar algo más que la igualdad y la desigualdad. PHP también te proporciona `>` (mayor que), `<` (menor que), `>=` (mayor que o igual a), y `<=` (menor que o igual a) con los que jugar. El Ejemplo 4-15 muestra cómo se utilizan.

Ejemplo 4-15. Los cuatro operadores de comparación

```
<?php
  $a = 2; $b = 3;

  if ($a > $b)   echo "$a is greater than $b<br>";
  if ($a < $b)   echo "$a is less than $b<br>";
  if ($a >= $b) echo "$a is greater than or equal to $b<br>";
  if ($a <= $b) echo "$a is less than or equal to $b<br>";
?>
```

En este ejemplo, en el que `$a` es `2` y `$b` es `3`, se obtiene la salida siguiente:

```
2 is less than 3
2 is less than or equal to 3
```

Puedes probar este ejemplo alterando los valores de `$a` y `$b`, para ver los resultados. Intenta asignar a las variables el mismo valor y verás lo que pasa.

Operadores lógicos

Los operadores lógicos producen resultados verdaderos o falsos y, por lo tanto, también se conocen como *operadores booleanos*. Hay cuatro tipos (ver la Tabla 4-4).

Tabla 4-4. Los operadores lógicos

Operador lógico	Descripción
`AND`	`TRUE` si los dos operandos son `TRUE`
`OR`	`TRUE` si cualquier operando es `TRUE`
`XOR`	`TRUE` si uno de los dos operandos es `TRUE`
`! (NOT)`	`TRUE` si el operando es `FALSE`, o `FALSE` si el operando es `TRUE`

Puedes ver cómo se utilizan estos operadores en el Ejemplo 4-16. Observa que el símbolo `!` lo utiliza PHP en lugar de `NOT`. Además, los operadores pueden escribirse en mayúsculas o minúsculas.

Ejemplo 4-16. Uso de los operadores lógicos

```
<?php
  $a = 1; $b = 0;

  echo ($a AND $b) . "<br>";
  echo ($a or $b)  . "<br>";
  echo ($a XOR $b) . "<br>";
  echo !$a         . "<br>";
?>
```

Línea por línea, este ejemplo da como resultado nada, 1, 1, y nada, lo que significa que solo la segunda y tercera declaraciones `echo` se evalúan como `TRUE`. (Recuerda que `NULL` [o nada] representa el valor `FALSE`). Esto se debe a que la sentencia `AND` requiere que los dos operandos sean `TRUE` si va a devolver un valor `TRUE`, mientras que la cuarta sentencia realiza un `NOT` sobre el valor de `$a` y lo convierte de `TRUE` (un valor de `1`) a `FALSE`. Si deseas experimentar con estos operadores, prueba el código cambiando los valores de `1` y `0` para `$a` y `$b`.

Al codificar, recuerda que `AND` y `OR` tienen menor prioridad que las otras versiones de los operadores, `&&` y `||`.

El operador `OR` puede causar problemas involuntarios en las declaraciones `if`, porque el segundo operando no se evaluará si el primero se evalúa como `TRUE`. En el Ejemplo 4-17, nunca se llamará a la función `getnext` si `$finished` tiene el valor de `1`.

Ejemplo 4-17. Declaración que utiliza el operador OR

```
<?php
  if ($finished == 1 OR getnext() == 1) exit;
?>
```

Si necesitas que se llame a `getnext` en cada declaración `if`, podrías reescribir el código como se ha hecho en el Ejemplo 4-18.

Ejemplo 4-18. La sentencia if...OR modificada para asegurar que se llama a getnext

```
<?php
  $gn = getnext();

  if ($finished == 1 OR $gn == 1) exit;
?>
```

En este caso, el código ejecuta la función `getnext` y almacena el valor devuelto en `$gn` antes de ejecutar la sentencia `if`.

Otra solución es cambiar las dos cláusulas para tener la seguridad de que `getnext` se ejecuta, ya que entonces aparecerá la primera en la expresión.

La Tabla 4-5 muestra todas las variaciones posibles del uso de los operadores lógicos. También debes tener en cuenta que `!TRUE` es igual a `FALSE`, y `!FALSE` es igual a `TRUE`.

Tabla 4-5. Todas las posibles expresiones lógicas en PHP

Entradas		Operadores y resultados		
a	**b**	**AND**	**OR**	**XOR**
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

Condicionales

Los *condicionales* alteran el flujo del programa. Te permiten hacer preguntas sobre ciertas cosas y responder a las respuestas que recibes, de diferentes maneras. Los condicionales son fundamentales para crear páginas web dinámicas (que es el objetivo de usar PHP en primer lugar) porque facilitan la presentación de diferentes resultados cada vez que se visualiza una página.

Presentaré tres condicionales básicos en esta sección: la declaración `if`, la declaración `switch` y el operador `?`. Además, los condicionales de bucle (a los que llegaremos en breve) ejecutan el código una y otra vez hasta que se cumpla una condición.

La declaración if

Una forma de pensar sobre el flujo de programas es imaginarlo como una carretera de un solo carril por la que vas conduciendo. Es más o menos una línea recta, pero de vez en cuando encontrarás varias señales que te dirán a dónde ir.

En el caso de una sentencia `if`, podrías imaginar que te encuentras con una señal de desvío que tienes que seguir si cierta condición es `TRUE`. Si es así, vas conduciendo y sigues el desvío hasta que regresas a la carretera principal y luego continúas tu camino siguiendo la ruta original. O, si la condición no es `TRUE`, ignoras el desvío y sigues conduciendo (ver la Figura 4-1).

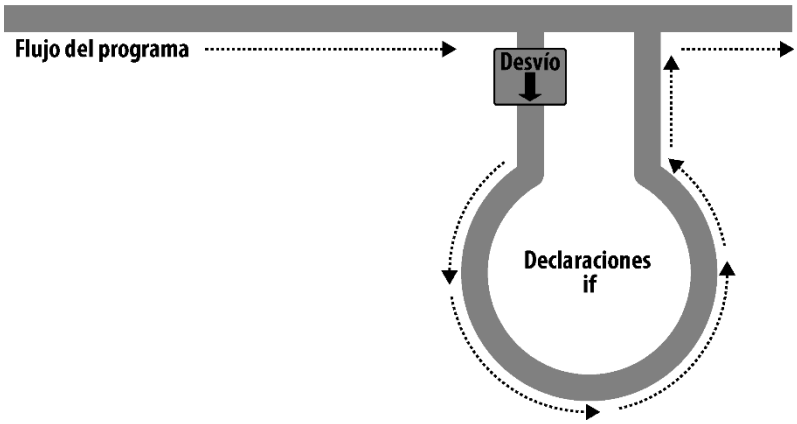

Figura 4-1. El flujo de un programa es como una carretera con un solo carril

El contenido de la condición `if` puede ser cualquier expresión PHP válida, incluidas pruebas de igualdad, expresiones de comparación, pruebas para `0` y `NULL`, e incluso funciones (ya sean funciones integradas o alguna que tú escribas).

Las acciones a llevar a cabo cuando una condición `if` es `TRUE` se colocan generalmente dentro de llaves ({ }). Puedes ignorar las llaves si solo tienes que ejecutar una sola declaración, pero si siempre las usas, evitarás tener que descubrir errores difíciles de rastrear, como cuando añades una línea extra a una condición y no se evalúa debido a que no has utilizado las llaves.

Una célebre vulnerabilidad de seguridad conocida como fallo "goto fail" ha perseguido el código Secure Sockets Layer (SSL) en los productos de Apple durante muchos años porque un programador había olvidado utilizar las llaves en una declaración `if`, lo que provocaba que una función a veces informara de que la conexión se había realizado correctamente cuando en realidad puede que no fuera así. Esto permitía a un atacante malicioso obtener un certificado seguro para ser aceptado cuando debería haber sido rechazado. En caso de duda, coloca llaves que contengan las acciones de las declaraciones `if`.

Observa que para mayor brevedad y claridad, sin embargo, muchos de los ejemplos en este libro ignoran esta sugerencia y omiten las llaves en declaraciones individuales.

En el Ejemplo 4-19, imagina que es fin de mes, que has pagado todas tus facturas y estás realizando algún tipo de mantenimiento de cuenta bancaria.

Ejemplo 4-19. Declaración if con llaves

```
<?php
  if  ($bank_balance  <  100)
  {
    $money          = 1000;
    $bank_balance  += $money;
  }
?>
```

En este ejemplo, estás verificando tu saldo para ver si es menor de 100 dólares (o sea cual sea tu moneda). Si es así, te pagas 1000 dólares y luego lo añades al saldo. (¡Si ganar dinero fuera tan simple!).

Si el saldo en el banco es de 100 dólares o más, las declaraciones condicionales se ignoran y el flujo del programa pasa a la siguiente línea (no se muestra aquí).

En este libro, la apertura de llaves generalmente comienza en una nueva línea. A algunas personas les gusta colocar la primera llave a la derecha de la expresión condicional; otras inician una nueva línea con ella. Cualquiera de estas opciones es correcta porque PHP te permite establecer los espacios en blanco (espacios, saltos de página y tabulaciones) de la manera que elijas. Sin embargo, encontrarás que el código es más fácil de leer y depurar si sangras cada nivel de los condicionales con una tabulación.

La declaración else

A veces, cuando un condicional no es `TRUE`, es posible que no desees continuar con el la otra declaración entra en juego. Con ella, puedes configurar un segundo desvío en tu código del programa principal porque puede que quieras hacer otra cosa. Aquí es donde autopista, como se muestra en la Figura 4-2.

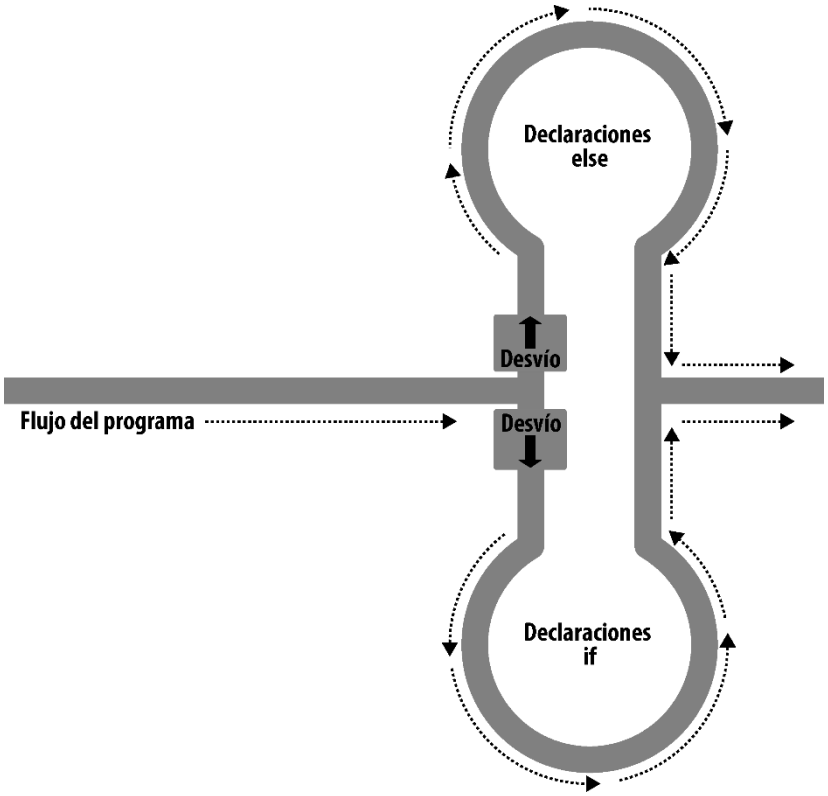

Figura 4-2. La autopista tiene ahora un desvío if y un desvío else

En una declaración `if...else`, la primera declaración condicional se ejecuta si la condición es `TRUE`. Pero si es `FALSE`, se ejecuta la segunda. Se *debe* ejecutar una de las dos opciones. Bajo ninguna circunstancia se pueden ejecutar ambas (o ninguna). El Ejemplo 4-20 muestra el uso de la estructura `if...else`.

Ejemplo 4-20. Declaración if...else con llaves

```
<?php
  If ($bank_balance  <  100)
  {
    $money          = 1000;
    $bank_balance  += $money;
  }
  else
  {
    $savings       += 50;
    $bank_balance  -= 50;
  }
?>
```

En este ejemplo, si has averiguado que tienes 100 dólares o más en el banco, se ejecuta la declaración `else` y coloca parte de este dinero en tu cuenta de ahorros.

Al igual que con las sentencias `if`, si `else` tiene solo una sentencia condicional, puedes optar por no utilizar las llaves. (Sin embargo, siempre se recomiendan las llaves. En primer lugar, hacen que el código sea más fácil de entender. En segundo lugar, te permiten añadir fácilmente más declaraciones a la bifurcación más tarde).

La declaración elseif

También hay ocasiones en las que quieres que ocurran una serie de posibilidades diferentes, según una secuencia de condiciones. Puedes lograrlo si utilizas la expresión `elseif`. Como puedes imaginar, es como una declaración `else`, excepto que colocas una expresión condicional adicional antes del código condicional `else`. En el Ejemplo 4-21, se puede ver una construcción completa `if...elseif...else`.

Ejemplo 4-21. Declaración if...elseif...else con llaves

```
<?php
  if ($bank_balance < 100)
  {
    $money          = 1000;
    $bank_balance  += $money;
  }
  elseif ($bank_balance > 200)
  {
    $savings       += 100;
    $bank_balance  -= 100;
  }
  else
  {
    $savings       += 50;
    $bank_balance  -= 50;
  }
?>
```

En el ejemplo, se ha insertado una sentencia `elseif` entre las sentencias `if` y `else`. Verifica si tu saldo bancario excede los 200 dólares y, si es así, decide que puedes ahorrar 100 dólares este mes.

Aunque estoy empezando a estirar la metáfora demasiado, puedes imaginarte esto como una serie de desvíos en varias direcciones (ver la Figura 4-3).

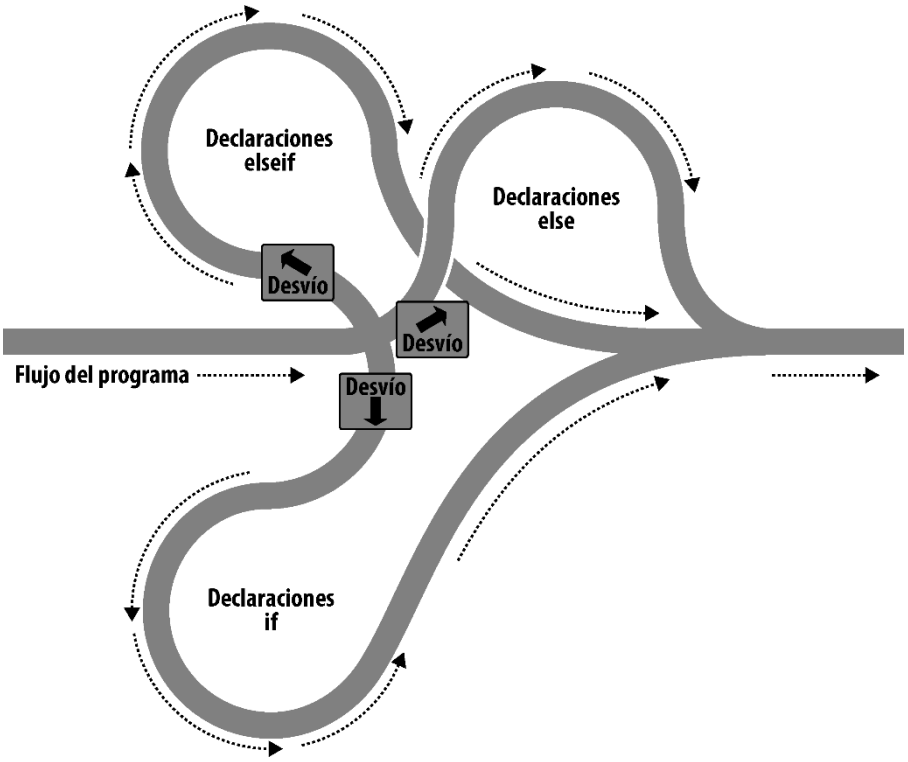

Figura 4-3. La autopista con los desvíos if, elseif y else

Una declaración `else` cierra bien una declaración `if...else` o una declaración `if...elseif...else`. Puedes omitir el `else` final si no es necesario, pero no puede aparecer `else` antes de un `elseif`; ni puede aparecer un `elseif` antes de una declaración `if`.

Puedes tener tantos `elseif` como quieras. Pero a medida que aumenta el número de `elseif`, probablemente sería mejor que consideraras la declaración `switch` si se ajusta a tus necesidades. Lo veremos a continuación.

La declaración switch

La declaración `switch` es útil cuando una variable, o el resultado de una expresión, puede tener varios valores, cada uno de los cuales debería desencadenar una actividad diferente. Por ejemplo, considera un menú basado en PHP que pasa una sola cadena al código del menú principal de acuerdo con lo que el usuario solicite. Digamos que las opciones son *Home* (Página principal), *About* (Acerca de), *News* (Noticias), *Login* (Inicio de sesión) y *Links* (Enlaces), y fijamos la variable `$page` para una de ellas, de acuerdo con las entradas que teclea el usuario.

Si escribimos el correspondiente código usando `if...elseif...else`, podría parecerse al Ejemplo 4-22.

Ejemplo 4-22. Declaración de varias líneas if...elseif...else

```
<?php
  if      ($page == "Home")   echo "You selected Home";
  elseif  ($page == "About")  echo "You selected About";
  elseif  ($page == "News")   echo "You selected News";
  elseif  ($page == "Login")  echo "You selected Login";
  elseif  ($page == "Links")  echo "You selected Links";
  else                        echo "Unrecognized selection";
?>
```

Si usamos una declaración `switch`, el código podría parecerse al del Ejemplo 4-23.

Ejemplo 4-23. La declaración switch

```
<?php
  switch ($page)
  {
    case "Home":
        echo "You selected Home";
        break;
    case "About":
        echo "You selected About";
        break;
    case "News":
        echo "You selected News";
        break;
    case "Login":
        echo "You selected Login";
        break;
    case "Links":
        echo "You selected Links";
        break;
  }
?>
```

Como puedes ver, `$page` solo se menciona una vez al principio de la declaración `switch`. A continuación, el comando `case` comprueba si hay coincidencias. Cuando se produce una, se ejecuta la sentencia condicional correspondiente. Por supuesto, en un programa real se necesitaría añadir aquí el código para mostrar o saltar a una página, en lugar de simplemente decirle al usuario lo que se ha seleccionado.

Con las declaraciones `switch`, no se utilizan llaves con los comandos `case`. En su lugar, comienzan con dos puntos y terminan con la declaración `break`. Sin embargo, la lista de casos de la declaración `switch` la encierran unas llaves.

Escape

Si deseas salir de la declaración `switch` porque se ha cumplido una condición, usa el comando `break`. Este comando le dice a PHP que salga de `switch` y salte a la siguiente declaración.

Si omitieras los comandos `break` del Ejemplo 4-23 y el caso de `Home` tuviera un valor `TRUE`, se ejecutarían entonces los cinco casos. O, si `$page` tuviera el valor `News`, a partir de entonces se ejecutarán todos los comandos `case`. La omisión deliberada permite cierta programación avanzada, pero por lo general siempre debes recordar emitir un comando `break` cada vez que un conjunto de condicionales `case` haya terminado de ejecutarse. De hecho, omitir la declaración `break` es un error muy común.

Acción por defecto

Un requisito típico en las sentencias `switch` es recurrir a una acción predeterminada si no se cumple ninguna de las condiciones `case`. Por ejemplo, en el caso del código del menú en el Ejemplo 4-23, podrías agregar el código del Ejemplo 4-24 inmediatamente antes de la llave final.

Ejemplo 4-24. Declaración por defecto para añadir al Ejemplo 4-23

```
default:
    echo "Unrecognized selection";
    break;
```

Esto reproduce el efecto de la declaración `else` en el Ejemplo 4-22.

Aunque aquí no se requiere un comando `break` porque el valor por defecto es la subexpresión final y el flujo del programa continuará automáticamente hasta la llave de cierre, si decides colocar la declaración `default` más arriba, definitivamente necesitaría un comando `break` para evitar que el flujo del programa continúe con las siguientes declaraciones. Generalmente, la práctica más segura es incluir siempre el comando `break`.

Sintaxis alternativa

Si lo prefieres, puedes sustituir la llave de apertura en una declaración `switch` con solo dos puntos y la llave de cierre con un comando `endswitch`, como en el Ejemplo 4-25. Sin embargo, este enfoque no se utiliza normalmente y se menciona aquí solo en caso de que lo encuentres en código de terceros.

Ejemplo 4-25. Sintaxis de declaración alternativa de switch

```
<?php
  switch ($page):
    case "Home":
        echo "You selected Home";
        break;
```

```
    // etc

    case "Links":
        echo "You selected Links"; break;
  endswitch;
?>
```

El operador ?

Una forma de evitar la verborrea de las declaraciones `if` y `else` es utilizar el operador ternario ?, más compacto, que no es muy usual, ya que requiere tres operandos en lugar de los dos típicos.

Encontramos este operador del que se hace una breve mención en el Capítulo 3, en la discusión sobre la diferencia entre las declaraciones `print` y `echo`, como un ejemplo de un tipo de operador que trabaja bien con `print`, pero no con `echo`.

Al operador `?` se le pasa una expresión que debe evaluar, junto con dos estados a ejecutar: uno para el caso en el que la expresión se evalúa a `TRUE`, el otro para cuando es `FALSE`. El Ejemplo 4-26 muestra el código que podemos usar para escribir una advertencia sobre el nivel de combustible de un coche en el salpicadero digital.

Ejemplo 4-26. Uso del operador ?

```
<?php
  echo $fuel <= 1 ? "Fill tank now" : "There's enough fuel";
?>
```

En esta declaración, si hay un galón o menos de combustible (en otras palabras, si el contenido de `$fuel` es menor o igual que 1), la cadena `Fill tank now` se devuelve a la anterior declaración `echo`. De otra manera lo que se devuelve es la cadena `There's enough fuel`. Se puede también asignar el valor devuelto por una declaración `?` a una variable (ver el Ejemplo 4-27).

Ejemplo 4-27. Asignación del resultado condicional de ? a una variable

```
<?php
$enough = $fuel <= 1 ? FALSE : TRUE;
?>
```

Aquí, a `$enough` se le asignará el valor `TRUE` solo cuando haya más de un galón de combustible; de lo contrario, se le asigna el valor `FALSE`.

Si el operador `?` te resulta confuso, siempre puedes acudir a las declaraciones `if`, pero deberías estar familiarizado con este operador, porque lo encontrarás en el código que hayan escrito otros desarrolladores. Puede ser difícil de leer, porque a menudo mezcla varios acontecimientos de la misma variable. Por ejemplo, un código como el siguiente es bastante conocido:

```
    $saved = $saved >= $new ? $saved : $new;
```

Si lo analizas cuidadosamente, puedes descubrir que el código hace lo siguiente:

```
$saved =                // Set the value of $saved to...
     $saved >= $new     // Check $saved against $new
  ?                     // Yes, comparison is true...
     $saved             // ... so assign the current value of $saved
  :                     // No, comparison is false...
     $new;              // ... so assign the value of $new
```

Es una forma resumida de hacer un seguimiento del valor más alto que aparece en un programa a medida que este se va ejecutando. Guardas el valor más alto en `$saved` y lo comparas con `$new` cada vez que obtienes un nuevo valor. Los programadores familiarizados con el operador `?` lo consideran más conveniente que la declaración `if` para comparaciones como la anterior. Cuando no se utiliza para escribir código compacto, se emplea generalmente para tomar decisiones en línea, como es el caso de probar si una variable está configurada antes de pasarla a una función.

Bucles

Una de las grandes cosas de los ordenadores es que pueden repetir tareas de cálculo rápida e incansablemente. A menudo puedes querer que un programa repita la misma secuencia de código una y otra vez hasta que suceda algo, como por ejemplo que un usuario introduzca un valor o que finalice su visita. Las estructuras de bucles de PHP proporcionan la forma perfecta para conseguir esto.

Para imaginarnos cómo funcionan, podemos ver la Figura 4-4. Es muy parecido a la metáfora de la autopista usada para ilustrar las declaraciones `if`, excepto que el desvío también tiene una sección de bucle de la que un vehículo (una vez que se ha incorporado) puede salir solo bajo las condiciones adecuadas del programa.

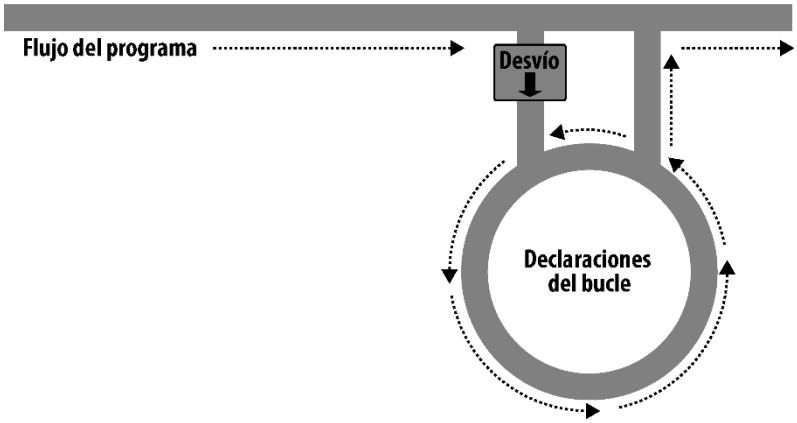

Figura 4-4. Podemos imaginar un bucle como parte de un programa de trazado de autopista

Bucles while

Podemos transformar el salpicadero digital del coche del Ejemplo 4-26 para que presente el resultado de un bucle que continuamente comprueba el nivel de combustible mientras conduces, mediante el bucle while (Ejemplo 4-28).

Ejemplo 4-28. Bucle while

```
<?php
  $fuel = 10;

  while ($fuel > 1)
  {
    // Keep driving...
    echo "There's enough fuel";
  }
?>
```

En realidad, es posible que prefieras mantener una luz verde encendida en lugar de tener como resultado un texto, pero lo importante es que cualquier indicación sobre el nivel de combustible se coloca dentro del bucle `while`. Si pruebas este ejemplo, ten en cuenta que seguirá imprimiendo la cadena hasta que hagas clic en el botón Stop en tu navegador.

Al igual que con las declaraciones `if`, notarás que se necesitan las llaves para incluir en ellas las declaraciones del bucle `while`, a menos que solo haya una.

Otro ejemplo de bucle `while`, que presenta la tabla de multiplicar del 12, se puede ver en el Ejemplo 4-29.

Ejemplo 4-29. Bucle while para presentar la tabla de multiplicar del 12

```
<?php
  $count = 1;

  while ($count <= 12)
  {
    echo "$count times 12 is " . $count * 12 . "<br>";
    ++$count;
  }
?>
```

Aquí la variable `$count` se inicializa a 1, y luego comienza un bucle `while` con la expresión comparativa `$count <= 12`. Este bucle continuará ejecutándose hasta que la variable sea mayor que 12. La salida de este código es la siguiente:

```
1 times 12 is 12
2 times 12 is 24
3 times 12 is 36
etc...
```

Dentro del bucle, se imprime una cadena junto con el valor de `$count` multiplicado por `12`.

Para mayor claridad, le sigue una etiqueta `<br>` para forzar una nueva línea. Entonces se incrementa `$count`, y le sigue la llave de cierre que le dice a PHP que regrese al inicio del bucle.

En este punto, se vuelve a probar `$count` para ver si es mayor de `12`. No lo es, pero ahora tiene el valor `2`, y después de ejecutar el bucle otras `11` veces, tendrá el valor `13`. Cuando eso sucede, se salta el código contenido en el bucle `while` y se ejecuta el código que sigue al bucle, que, en este caso, es el final del programa.

Si la declaración `++$count` (que podría haber sido `$count++`) no hubiera estado ahí, este bucle sería como el primero de esta sección. Nunca terminaría, y solo se imprimiría el resultado de `1 * 12` una y otra vez.

Pero hay una manera mucho más ordenada de escribir este bucle, que creo que te gustará. Echa un vistazo al Ejemplo 4-30.

Ejemplo 4-30. Una versión abreviada del Ejemplo 4-29

```
<?php
  $count = 0;

  while (++$count <= 12)
    echo "$count times 12 is " . $count * 12 . "<br>";
?>
```

En este ejemplo, era posible mover la declaración `++$count` desde las declaraciones que forman parte del bucle `while` a la expresión condicional del bucle. Lo que sucede ahora es que PHP encuentra la variable `$count` al inicio de cada iteración del bucle y, al notar que está precedido por el operador incremento, primero incrementa la variable y solo entonces lo compara con el valor `12`. Por lo tanto, puedes ver que `$count` ahora tiene que inicializarse a `0`, no a `1`, porque se incrementa tan pronto como entra en el bucle. Si mantuviera la inicialización en `1`, solo se obtendrían resultados entre `2` y `12`.

Bucles do...while

Una ligera variación del bucle `while` es el bucle `do...while`, que se utiliza cuando se desea que un bloque de código se ejecute al menos una vez y el condicional se sitúa después de ese bloque. El Ejemplo 4-31 muestra una versión modificada del código para la tabla del 12 que utiliza dicho bucle.

Ejemplo 4-31. Bucle do...while para presentar la tabla de multiplicar del 12

```
<?php
  $count = 1;
  do
    echo "$count times 12 is " . $count * 12 . "<br>";
  while (++$count <= 12);
?>
```

Observa cómo hemos vuelto a inicializar `$count` a `1` (en vez de `0`) debido a que la instrucción `echo` del ciclo se ejecuta antes de que tengamos la oportunidad de incrementar la variable. Aparte de eso, sin embargo, el código es bastante similar.

Por supuesto, si hay más de una declaración en el bucle `do...while`, recuerda usar llaves, como en el Ejemplo 4-32.

Ejemplo 4-32. Ampliación del Ejemplo 4-31, por lo que es necesario usar llaves

```
<?php
  $count = 1;

  do {
    echo "$count times 12 is " . $count * 12;
    echo "<br>";
  } while (++$count <= 12);
?>
```

Bucles for

La última clase de declaración de bucle, la de bucle `for`, es también la más potente, ya que combina la capacidad de configurar variables cuando se entra en el bucle, probar condiciones mientras se iteran bucles y modificar variables después de cada iteración.

El Ejemplo 4-33 muestra cómo escribir el programa de la tabla de multiplicar con un bucle `for`.

Ejemplo 4-33. Salida de la tabla de multiplicar del 12 con un bucle for

```
<?php
  for ($count = 1 ; $count <= 12 ; ++$count)
    echo "$count times 12 is " . $count * 12 . "<br>";
?>
```

¿Has visto como se ha reducido el código a una sola declaración `for` que contiene una única declaración condicional? Lo que ocurre es lo siguiente. Cada declaración `for` tiene tres parámetros:

- Una expresión de inicialización
- Una expresión condicional
- Una expresión de modificación

Se utiliza el punto y coma para separarlas, como lo siguiente: `for (expr1; expr2; expr3)`. Al iniciar la primera iteración del bucle, se ejecuta la expresión de inicialización. En el caso del código de la tabla de multiplicar, `$count` se inicializa al valor `1`. Entonces, cada vez que se ejecuta el bucle, se prueba la expresión de condición (en este caso, `$count <= 12`), y el bucle se introduce solo si la condición es `TRUE`. Por último, al final de cada iteración, se ejecuta la expresión de modificación. En el caso del código de la tabla de multiplicar, se incrementa la variable `$count`.

Toda esta estructura elimina claramente la necesidad de que el bucle contenga los elementos de control, y deja en el mismo solo las declaraciones que deseas que este realice.

Recuerda usar llaves con un bucle `for` si este va a contener más de una declaración, como en el Ejemplo 4-34.

Ejemplo 4-34. El bucle for loop del Ejemplo 4-33 con llaves añadidas

```
<?php
  for ($count = 1 ; $count <= 12 ; ++$count)
  {
    echo "$count times 12 is " . $count * 12;
    echo "<br>";
  }
?>
```

Vamos a comparar cuándo usar los bucles `for` y `while`. El diseño del bucle `for` gira alrededor de un único valor que cambia regularmente. Por lo general, tienes un valor que se incrementa, como cuando te pasan una lista de opciones de usuario y deseas procesar una por una. Pero para ello puedes transformar la variable como quieras. Una forma más compleja de la sentencia `for` incluso te permite realizar múltiples operaciones en cada uno de los tres parámetros:

```
for ($i = 1, $j = 1 ; $i + $j < 10 ; $i++ , $j++)
{
// ...
}
```

Sin embargo, esto es complicado y no es recomendable para usuarios que lo utilizan por primera vez. La clave es distinguir entre coma y punto y coma. Los tres parámetros deben estar separados por punto y coma. Dentro de cada parámetro, se pueden separar múltiples declaraciones por comas.

Así, en el ejemplo anterior, los parámetros primero y tercero contienen cada uno dos declaraciones:

```
$i = 1, $j = 1  // Initialize $i and $j
$i + $j < 10    // Terminating condition
$i++ , $j++     // Modify $i and $j at the end of each iteration
```

Lo principal a tener en cuenta de este ejemplo es que debes separar las tres secciones de parámetros con punto y coma, no comas (las cuales se deben usar solo para separar declaraciones dentro de la sección de un parámetro).

Entonces, ¿cuándo es más apropiada una declaración `while` que una declaración `for`? Cuando tu condición no depende de que la variable experimente un cambio simple y de forma regular. Por ejemplo, si deseas comprobar si hay alguna entrada o error especial y finalizar el bucle cuando esto se produzca, usa una declaración `while`.

Salida del bucle

De la misma manera que viste cómo salir de una declaración `switch`, también puedes salir de un bucle `for` (o de cualquier bucle) con el mismo comando `break`. Este paso puede ser necesario cuando, por ejemplo, una de las declaraciones devuelve un error y el bucle no puede continuar ejecutándose de forma segura.

Un caso en el que esto podría ocurrir es cuando al escribir un archivo se devuelve un error, posiblemente porque el disco esté lleno (ver Ejemplo 4-35).

Ejemplo 4-35. Escritura de un archivo usando un bucle for con captura de errores

```
<?php
  $fp = fopen("text.txt", 'wb');

  for ($j = 0 ; $j < 100 ; ++$j)
  {
    $written = fwrite($fp, "data");

    if ($written == FALSE) break;
  }
  fclose($fp);
?>
```

Este es el fragmento de código más complicado que hemos visto hasta ahora, pero estás preparado para ello. Examinaremos los comandos de manejo de archivos en el Capítulo 7, pero por ahora todo lo que debes saber es que la primera línea abre el archivo *text.txt* para escribir en modo binario, y luego devuelve un puntero del archivo a la variable `$fp`, que se usa más tarde para referirse al archivo abierto.

El bucle se itera 100 veces (de 0 a 99), escribiendo los `data` (datos) de la cadena en el archivo. Después de cada escritura, la función `fwrite` asigna un valor a la variable `$written` que representa el número de caracteres correctamente escritos. Pero si hay un error, la función `fwrite` asigna el valor `FALSE`.

El comportamiento de `fwrite` hace que sea fácil para el código comprobar la variable `$written` para ver si se ha puesto en `FALSE` y, si es así, salir del bucle con la siguiente sentencia que cierra el archivo.

Si buscas mejorar el código, puedes simplificar la línea:

```
if ($written == FALSE) break;
```

con el operador `NOT`, así:

```
if (!$written) break;
```

De hecho, el par de declaraciones del bucle interno se pueden reducir a una sola declaración:

```
if (!fwrite($fp, "data")) break;
```

En otras palabras, puedes eliminar la variable `$written`, porque existía solo para verificar el valor devuelto por `fwrite`. En su lugar, puedes probar el valor de retorno directamente.

El comando `break` es aún más potente de lo que se podría pensar porque, si tienes el código anidado en más de una capa de la que necesitas salir, puedes escribir un número a continuación del comando `break` para indicar de cuántos niveles hay que salir.

```
    break 2;
```

Declaración continue

La declaración `continue` es un poco como la declaración `break`, excepto que instruye a PHP para detener el procesamiento de la iteración en curso del bucle y pasar directamente a la siguiente iteración. Así que, en lugar de romper todo el bucle, PHP solo sale de la iteración en curso.

Este enfoque puede ser útil en los casos en los que sabes que no tiene sentido continuar con la ejecución del bucle en curso y deseas guardar los ciclos del procesador o evitar un error que ocurre al moverse a lo largo de la siguiente iteración del bucle. En el Ejemplo 4-36, se utiliza la declaración `continue` para evitar que se emita un error de división entre cero cuando la variable `$j` tiene un valor de `0`.

Ejemplo 4-36. Captura de errores al dividir entre cero cuando se utiliza continue

```
<?php
  $j = 10;

  while ($j > -10)
  {
    $j--;

    if ($j == 0) continue;

    echo (10 / $j) . "<br>";
  }
?>
```

Para todos los valores de `$j` entre `10` y `–10`, con la excepción de `0`, se obtiene el resultado de calcular `10` dividido entre `$j`. Pero para el caso de que `$j` sea `0`, se emite la declaración `continue` y la ejecución salta inmediatamente a la siguiente iteración del bucle.

Conversión implícita y explícita

PHP es un lenguaje poco estructurado que te permite declarar una variable y su tipo simplemente usándola. También convierte automáticamente valores de un tipo a otro cuando se requiere. A esto se le llama *conversion implícita*.

Sin embargo, a veces la conversión implícita de PHP puede no ser lo que necesitas. En el Ejemplo 4-37, ten en cuenta que las entradas a la división son enteros. Por defecto, PHP convierte la salida a punto flotante para poder obtener un valor más preciso: 4.66 periódico.

Ejemplo 4-37. Esta expresión devuelve un número en punto flotante

```
<?php
  $a = 56;
  $b = 12;
  $c = $a / $b;

  echo $c;
?>
```

Pero, ¿y si hubiéramos querido que `$c` fuera un número entero? Hay varias maneras de conseguirlo, una de las cuales es forzar a que el resultado de `$a / $b` sea la conversión a un valor entero mediante el tipo de conversión a entero (`int`), así:

```
$c = (int) ($a / $b);
```

A esto se le llama conversión *explícita*. Observa que para tener la seguridad de que el valor de toda la expresión sea un número entero, colocamos la expresión entre paréntesis. De lo contrario, solo la variable `$a` se habría convertido a un número entero (una operación absurda, ya que la división por `$b` habría devuelto un número de punto flotante).

Puedes convertir explícitamente variables y literales a los tipos que se muestran en la Tabla 4-6.

Tabla 4-6. Tipos de conversión de PHP

Tipo de conversión	Descripción
`(int) (integer)`	Convierte a un entero al descartar la parte decimal
`(bool) (boolean)`	Convierte a booelano
`(float) (double) (real)`	Convierte a un número en punto flotante
`(string)`	Convierte a una cadena
`(array)`	Convierte a una matriz
`(object)`	Convierte a un objeto

Normalmente puedes evitar tener que usar una conversión llamando a una de las funciones integradas en PHP. Por ejemplo, para obtener un valor entero, se puede utilizar la función `intval`. Como con algunas otras secciones de este libro, esta sección está aquí principalmente para ayudarte a entender el código de terceros que puedas encontrar.

Enlaces dinámicos en PHP

Debido a que PHP es un lenguaje de programación, y el resultado puede ser completamente diferente para cada usuario, puede ocurrir que todo un sitio web se ejecute desde una sola página web PHP. Cada vez que el usuario hace clic en algo, los detalles se pueden enviar de vuelta a la misma página web, que decide qué hacer a continuación de acuerdo con las diferentes cookies y/u otros detalles de la sesión que pueda haber almacenados.

Pero aunque es posible construir un sitio web completo de esta manera, no es recomendable, porque tu código fuente crecerá y crecerá, y comenzará a ser difícil de manejar, ya que este tiene que tener en cuenta todas las acciones posibles que un usuario podría elegir.

En cambio, es mucho más sensato dividir el desarrollo del sitio web en diferentes partes. Por ejemplo, un proceso inconfundible es inscribirse en un sitio web, junto con todas las comprobaciones necesarias para validar una dirección de correo electrónico, determinar si un nombre de usuario ya está ocupado, etc.

Un segundo módulo podría ser el que inicia la sesión de los usuarios antes de darles paso a la parte principal del sitio web. Podrías tener un módulo de mensajes para que los usuarios pudieran tener la posibilidad de dejar comentarios, un módulo que contenga enlaces e información útil, otro para permitir la carga de imágenes, etc.

Siempre y cuando hayas creado una forma de rastrear al usuario a través de tu sitio web por medio de cookies o mediante variables de sesión (las cuales veremos detenidamente en capítulos posteriores), podrás dividir tu sitio web en secciones lógicas de código PHP, cada una de ellas autocontenida, y darte el gusto de tener un futuro mucho más fácil, desarrollando nuevas características y conservando las antiguas. Si tienes un equipo, diferentes personas pueden trabajar en diferentes módulos, de modo que cada programador necesitará aprender a fondo solo una parte del programa.

Enlaces dinámicos en acción

Una de las aplicaciones PHP más populares en la web hoy en día es la plataforma Wordpress de blogs (ver Figura 4-5). Como blogger o lector de blogs, es posible que te des cuenta de ello, pero a cada sección principal se le ha dado su propio archivo PHP principal y toda una serie de funciones genéricas y compartidas en archivos separados que se incluyen en las páginas principales de PHP en función de las necesidades.

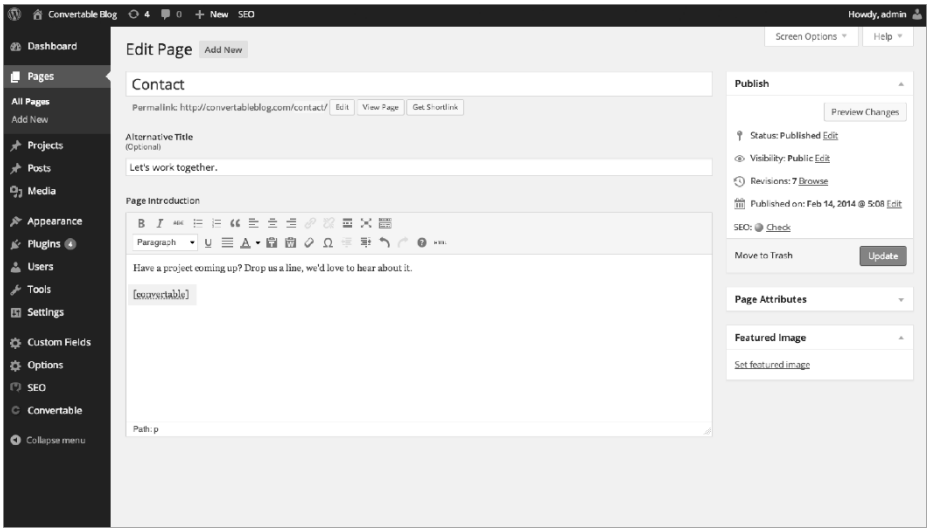

Figura 4-5. Panel de control de la plataforma de blogs WordPress

Toda la plataforma se mantiene, junto con un seguimiento de las secciones, en segundo plano, de modo que apenas te das cuenta de cuándo transitas de una subsección a otra. Por lo tanto, un desarrollador web que quiera modificar WordPress puede encontrar fácilmente el archivo que necesita en particular, lo modifica y lo prueba y depura sin tener que perder el tiempo con otras partes del programa. La próxima vez que uses WordPress, no pierdas de vista la barra de direcciones del navegador, especialmente si administras un blog, y te darás cuenta de algunos de los diferentes archivos PHP que utiliza.

Este capítulo ha cubierto bastante terreno, y a estas alturas ya deberías ser capaz de crear tus propios programas elementales PHP. Pero antes de hacerlo, y antes de proceder con el siguiente capítulo sobre funciones y objetos, puede que quieras probar tus nuevos conocimientos respondiendo a las siguientes preguntas.

Preguntas

1. ¿Qué valores subyacentes reales representan `TRUE` y `FALSE`?
2. ¿Cuáles son las dos formas de expresión más sencillas?
3. ¿Cuál es la diferencia entre operadores unarios, binarios y ternarios?
4. ¿Cuál es la mejor forma de forzar tu prioridad de operador?
5. ¿Qué se entiende por *asociatividad de operadores*?
6. ¿Cuándo utilizarías el operador `===` (identidad)?
7. Nombra los tres tipos de declaración condicional.

8. ¿Qué comando puedes usar para omitir la iteración en curso de un bucle y continuar a la siguiente?
9. ¿Por qué un bucle for es más potente que un bucle `while`?
10. ¿Cómo interpretan las declaraciones `if` y `while` las expresiones condicionales de diferentes tipos de datos?

Consulta "Respuestas del Capítulo 4" en la página 708 del Apéndice A para comprobar las respuestas a estas preguntas.