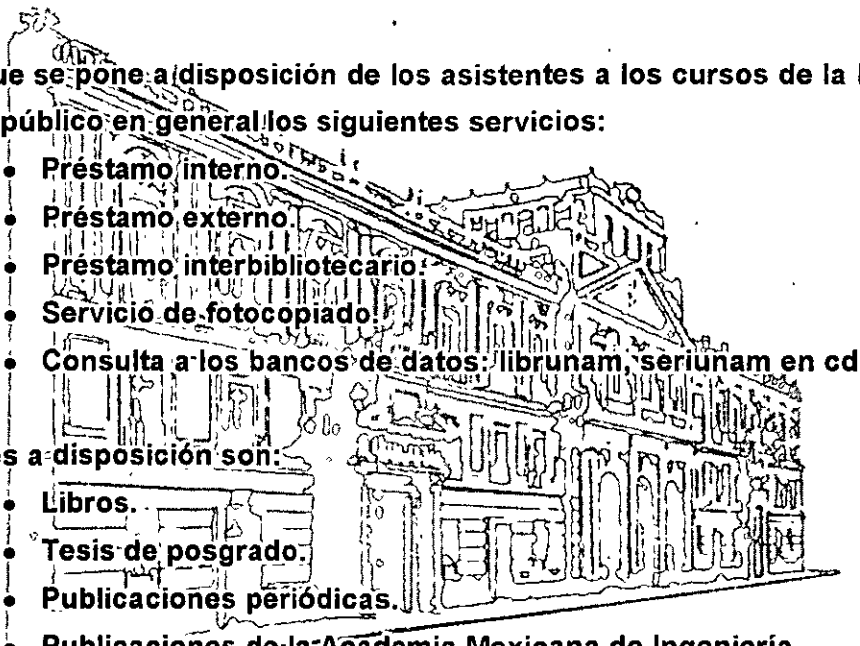
FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN
" ING. BRUNO MASCANZONI "

El Centro de Información y Documentación Ing. Bruno Mascanzoni tiene por objetivo satisfacer las necesidades de actualización y proporcionar una adecuada información que permita a los ingenieros, profesores y alumnos estar al tanto del estado actual del conocimiento sobre temas específicos, enfatizando las investigaciones de vanguardia de los campos de la ingeniería, tanto nacionales como extranjeras.



Es por ello que se pone a disposición de los asistentes a los cursos de la DECFI, así como del público en general los siguientes servicios:

- **Préstamo interno.**
- **Préstamo externo.**
- **Préstamo interbibliotecario.**
- **Servicio de fotocopiado.**
- **Consulta a los bancos de datos: librunam, seriunam en cd-rom.**

Los materiales a disposición son:

- **Libros.**
- **Tesis de posgrado.**
- **Publicaciones periódicas.**
- **Publicaciones de la Academia Mexicana de Ingeniería.**
- **Notas de los cursos que se han impartido de 1988 a la fecha.**

En las áreas de ingeniería industrial, civil, electrónica, ciencias de la tierra, computación y, mecánica y eléctrica.

El CID se encuentra ubicado en el mezzanine del Palacio de Minería, lado oriente.

El horario de servicio es de 10:00 a 14:30 y 16:00 a 17:30 de lunes a viernes.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

A LOS ASISTENTES A LOS CURSOS

Las autoridades de la Facultad de Ingeniería, por conducto del jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia solamente a los alumnos que tengan un mínimo de 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el periodo de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

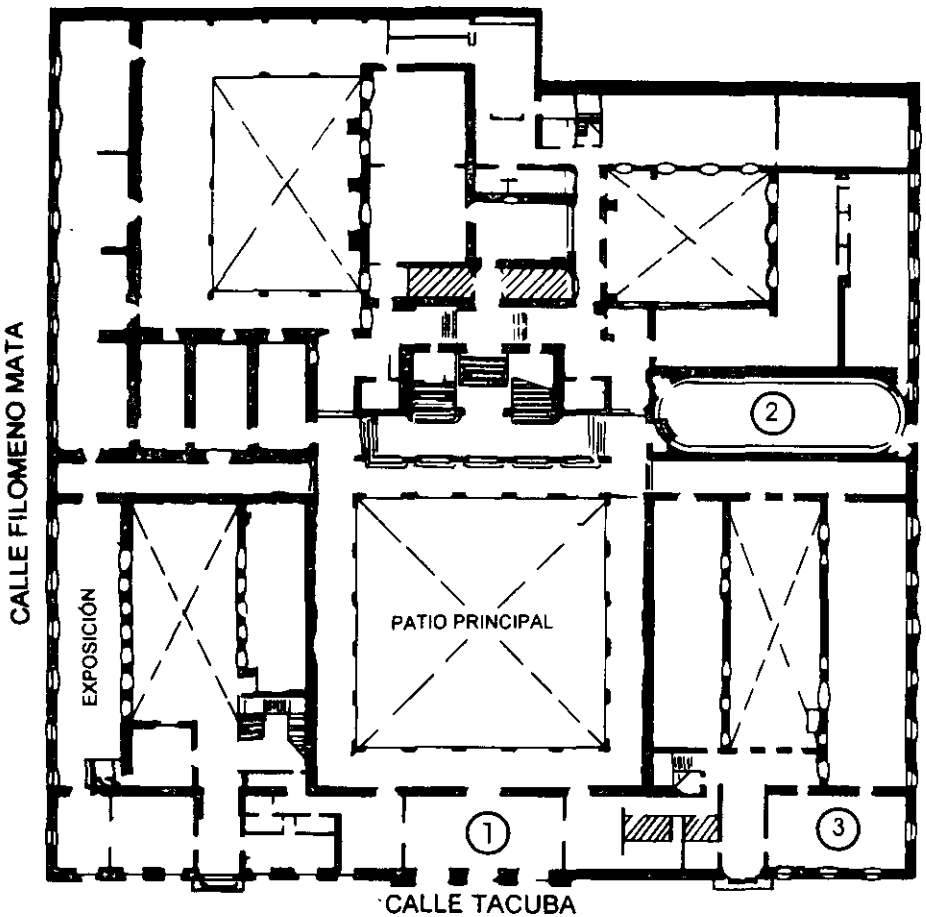
Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

Atentamente

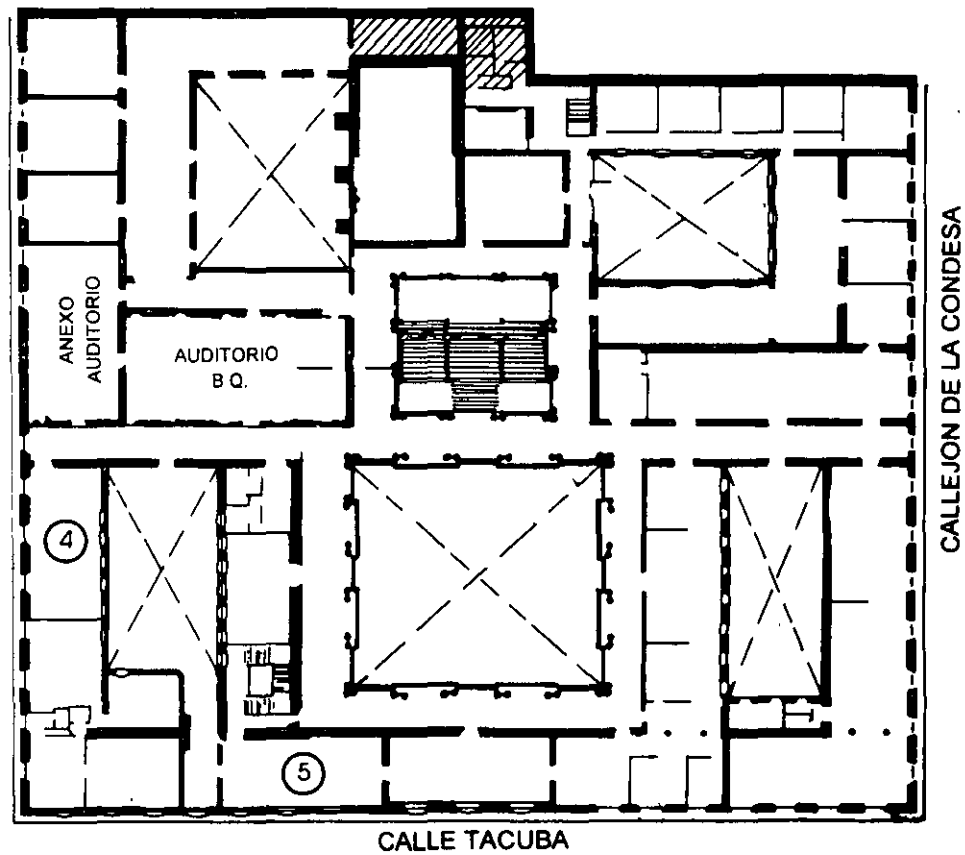
División de Educación Continua.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D F. APDO. Postal M-2285
Teléfonos: 5512-8955 5512-5121 5521-7335 5521-1987 Fax 5510-0573 5521-4021 AL 25

PALACIO DE MINERIA

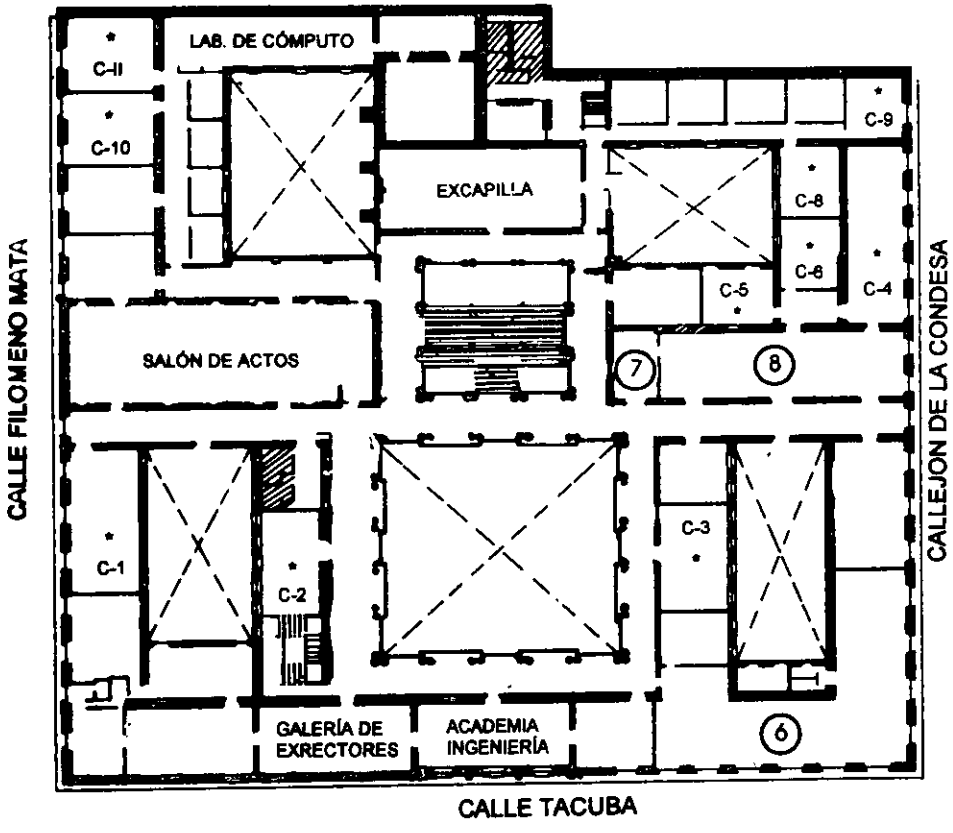


PLANTA BAJA



MEZZANINNE

PALACIO DE MINERIA



Ier. PISO

GUÍA DE LOCALIZACIÓN

1. ACCESO
2. BIBLIOTECA HISTÓRICA
3. LIBRERÍA UNAM
4. CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN "ING. BRUNO MASCANZONI"
5. PROGRAMA DE APOYO A LA TITULACIÓN
6. OFICINAS GENERALES
7. ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA
8. SALA DE DESCANSO

SANITARIOS

* AULAS

DIVISIÓN DE EDUCACIÓN CONTINUA
FACULTAD DE INGENIERÍA U.N.A.M.
CURSOS ABIERTOS

DIVISIÓN DE EDUCACIÓN CONTINUA
DEC

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

MATERIAL DIDACTICO DEL CURSO

IMPLEMENTACION DE BASES DE DATOS EN PÁGINAS WEB CON PHP Y POSTGRESQL

5 AL 16 DE NOVIEMBRE DE 2001

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 5512-8955 5512-5121 5521-7335 5521-1987 Fax 5510-0573 5521-4021 AL 25

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA, UNAM
CURSOS ABIERTOS

CURSO: CC072 IMPLEMENTACION DE BASES DE DATOS EN PAGINAS WEB....
FECHA: 5 al 16 de noviembre de 2001

EVALUACIÓN DEL PERSONAL DOCENTE

(ESCALA DE EVALUACIÓN: 1 A 10)

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACIÓN CON EL ASISTENTE	PUNTUALIDAD
Jonathan Mora Cuevas				
Hugo Alejandro Reyes Sáenz				

Promedio _____

EVALUACIÓN DE LA ENSEÑANZA

CONCEPTO	CALIF
ORGANIZACIÓN Y DESARROLLO DEL CURSO	
GRADO DE PROFUNDIDAD DEL CURSO	
ACTUALIZACIÓN DEL CURSO	
APLICACIÓN PRACTICA DEL CURSO	

Promedio _____

EVALUACIÓN DEL CURSO

CONCEPTO	CALIF.
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDÁCTICO UTILIZADO	

Promedio _____

Evaluación total del curso_____

Continúa...2

1. ¿Le agradó su estancia en la División de Educación Continua?

SI ☐ NO ☐

Si indica que "NO" diga porqué: _____

2. Medio a través del cual se enteró del curso:

Periódico *La Jornada*	
Folleto anual	
Folleto del curso	
Gaceta UNAM	
Revistas técnicas	
Otro medio (Indique cuál)	

3. ¿Qué cambios sugeriría al curso para mejorarlo?

_____

_____

_____

_____

_____

_____

4. ¿Recomendaría el curso a otra(s) persona(s) ?

SI ☐ NO ☐

5.¿Qué cursos sugiere que imparta la División de Educación Continua?

_____

_____

_____

_____

_____

_____

6. Otras sugerencias:

_____

_____

_____

_____

_____

PHP y PostgreSQL

Índice

Introducción

Aunque PHP y ASP tienen básicamente la misma funcionalidad, su sintaxis es completamente distinta y su semántica también presenta diferencias, por lo que aunque lo que podamos hacer con ASP también lo podamos hacer en PHP (y viceversa), la adaptación de uno a otro lenguaje puede resultar algo complicada al principio.

ASP ("Active Server Pages") es, más que un lenguaje de programación en sí, una tecnología que permite insertar en una página HTML código que se ejecutará en el servidor. Este código puede ser Java, JavaScript o, más comúnmente, VBScript. Por lo tanto cuando en este curso hagamos referencia al "lenguaje ASP", y no a la tecnología, nos referiremos a VBScript.

Por su parte, PHP ("PHP: Hypertext Preprocessor") provee de una tecnología similar a ASP para insertar código en las páginas HTML, pero PHP además de la tecnología es el lenguaje. En PHP sólo podemos incluir código en un lenguaje, PHP, que será el lenguaje que vamos a estudiar en el presente curso.

PHP es un lenguaje interpretado basado principalmente en C, C++ y Java, con los que comparte prácticamente toda su sintaxis y semántica, y aporta también algunas características de lenguajes interpretados como Perl y Bash. Debido a esto, la curva de aprendizaje para programadores que ya conozcan estos lenguajes es muy suave, prácticamente pueden sentarse delante del ordenador y comenzar a escribir código.

En cuanto a la tecnología detrás de PHP, ya en la versión 3.0 el intérprete de PHP era bastante más rápido que los intérpretes existentes de ASP, lo que junto con su buena integración con el servidor HTTP Apache y su capacidad de acceder a unos 20 sistemas de Bases de Datos distintos, lo ha convertido en un fuerte competidor frente a las "soluciones" de Microsoft. Con la versión 4.0 de PHP que ha visto la luz hace pocos meses la situación ha mejorado todavía más: el intérprete es más rápido (hasta 12 veces más rápido que el de la versión 3.0); se ha perfeccionado la integración de PHP con otros servidores además de Apache, entre otros IIS; y se ha modularizado todo el diseño interno, entre otras cosas independizado el intérprete del lenguaje PHP (Zend) del módulo de comunicación con el servidor, con lo que a partir de ahora es posible utilizar PHP como lenguaje interpretado en cualquier otro proyecto (ya se está trabajando para utilizar PHP como lenguaje para procedimientos en MySQL.).

1.1 ¿Qué es PHP?

PHP (acronimo de "PHP: Hypertext Preprocessor") es un lenguaje interpretado de alto nivel embebido en páginas HTML y ejecutado en el servidor.

1.2 ¿Qué se puede hacer con PHP?

Al nivel más básico, PHP puede hacer cualquier cosa que se pueda hacer con un script CGI, como procesar la información de formularios, generar páginas con contenidos dinámicos, o mandar y recibir cookies.

Quizas la característica más potente y destacable de PHP es su soporte para una gran cantidad de bases de datos. Escribir un interfaz via web para una base de datos es una tarea simple con PHP. Las siguientes bases de datos están soportadas actualmente:

Adabas D	Ingres	Oracle (OCI7 and OCI8)
dBase	InterBase	PostgreSQL
Empress	FrontBase	Solid
FilePro	mSQL	Sybase
IBM DB2	MySQL	Velocis
Informix	ODBC	Unix dbm

PHP también soporta el uso de otros servicios que usen protocolos como IMAP, SNMP, NNTP, POP3, HTTP y derivados. Tambien se pueden abrir sockets de red directos (raw sockets) e interactuar con otros protocolos.

1.3 Historia

PHP fue concebido en otono de 1994 por Rasmus Lerdorf. Las primeras versiones no distribuidas al público fueron usadas en un sus páginas web para mantener un control sobre quien consultaba su currículum. La primera versión disponible para el público a principios de 1995 fue conocida como "Herramientas para paginas web personales" (Personal Home Page Tools). Consistian en un analizador sintáctico muy simple que solo entendia unas cuantas macros y una serie de utilidades comunes en las páginas web de entonces, un libro de visitas, un contador y otras pequenas cosas. El analizador sintactico fue reescrito a mediados de 1995 y fue nombrado PHP/FI version 2. FI viene de otro programa que Rasmus habia escrito y que procesaba los datos de formularios. Asi que combinó las "Herramientas para paginas web personales", el "intérprete de formularios", anadio soporte para mSQL y PHP/FI vio la luz. PHP/FI crecio a gran velocidad y la gente empezó a contribuir en el código.
Es difícil dar estadísticas exactas, pero se estima que a finales de 1996 PHP/FI se estaba usando al menos en 15.000 páginas web alrededor del mundo. A mediados de 1997 este número habia crecido a mas de 50.000. A mediados de 1997 el desarrollo del proyecto sufrio

un profundo cambio, dejó de ser un proyecto personal de Rasmus, al cual habian ayudado un grupo de usuarios y se convirtio en un proyecto de grupo mucho más organizado. El analizador sintáctico se reescribio desde el principio por Zeev Suraski y Andi Gutmans y este nuevo analizador establecio las bases para PHP versión 3. Gran cantidad de codigo de PHP/FI fue portado a PHP3 y otra gran cantidad fue escrito complétamente de nuevo.
Hoy en día (finales 1999), tanto PHP/FI como PHP3 se distribuyen en un gran número de productos comerciales tales como el servidor web "C2's StrongHold" y Redhat Linux. Una estimación conservativa basada en estadísticas de NetCraft (ver también Estudio de NetCraft sobre servidores web), es que más de 1.000.000 de servidores alrededor del mundo usan PHP. Para hacernos una idea, este número es mayor que el número de servidores que utilizan el "Netscape's Enterprise server" en Internet.
A la vez que todo esto está pasando, el trabajo de desarrollo de la próxima generación de PHP está en marcha. Esta versión utiliza el potente motor de scripts Zend para proporcionar altas prestaciones, así como soporta otros servidores web, además de apache, que corren PHP como módulo nativo.

2. PHP como lenguaje de programación

2.1 Extensión de los ficheros

La primera diferencia entre ASP y PHP viene a la hora de dar nombre a los ficheros. La extensión es importante ya que el servidor HTTP (en nuestro caso, generalmente Apache) decide si debe pasárselo al procesador de PHP o no en función de esta extensión.

Las extensiones que indican al servidor HTTP que el fichero contiene código PHP que debe ser procesado son:

- .php3 Indica código PHP 3.x.
- .php4 Indica código PHP 4.x.
- .php Indica código PHP. Preferiremos esta extensión por ser más genérica.
- .phtml Actualmente en desuso.

Aunque haya extensiones diferentes para PHP 3 y 4, no tienen efecto en el intérprete que las procesará IGUAL. En general, PHP 4 es compatible con 3, salvo unas pocas excepciones. Este curso se centra en PHP 4, así que no analizaremos estos casos de incompatibilidad con las versiones anteriores.

2.2 Delimitadores de código

En PHP disponemos de cuatro opciones para delimitar el código:

```
<? echo 'Primer método de delimitar código PHP'; ?>

<?php echo 'Segundo método, el más usado'; ?>

<script language="php">
echo 'Algunos editores (como el FrontPage) Sólo entienden este método';
</script>

<% echo 'Método de compatibilidad con ASP'; %>
```

Los métodos primero y cuarto no siempre estarán disponibles, ya que se pueden activar o desactivar al compilar el intérprete de PHP. En general, preferiremos usar el segundo método.

2.3 Comentarios

En PHP hay tres formas de introducir comentarios en el código, frente al ' de ASP:

```
/* Comentarios estilo C.
 * Pueden extenderse durante varias líneas.
 */
```

```
// Comentarios estilo C++. Cubren hasta el final de la línea.

# Comentarios estilo Bash/Perl. Cubren hasta el fin de línea.
```

Hay que hacer notar que los comentarios de una línea cubren hasta final de línea O HASTA EL FIN DEL BLOQUE PHP ('?>' o el que corresponda.)

2.4 Variables

Las variables son una parte fundamental de todo lenguaje de programación. En ellas se almacenan valores con los que se puede operar, se pueden comparar entre sí y se puede hacer variar el flujo del programa en función de su valor. Vamos a ver cómo trabajar con variables en PHP.

Al contrario que en la mayoría de lenguajes de programación, y al igual que en ASP, en PHP NO hace falta declarar las variables antes de usarlas: tras la primera aparición en el código quedan declaradas. Como diferencia frente a ASP, en PHP todas las variables llevan delante el signo del dólar '$'. Ej. :

```
$var_1 = 123;
$var_2 = 'hola';
$var_3 = $var_1 * 2;
```

En PHP, al igual que en ASP, una variable se asocia a un contenido del tipo que sea, estando el tipo indicado en el contenido, no en la variable en sí. Visto de otra forma, una misma variable se puede reutilizar asignándole a lo largo del tiempo datos de distinto tipo. Por ejemplo:

```
$mi_variable = 'Inicializamos como una cadena de texto';
$mi_variable = 3;                    // Ahora es un entero.
$mi_variable = 3.14 * $mi_variable; // Ahora un float.
$mi_variable = new MiClase();        // Ahora un objeto.
```

El tipo de una variable nos va a importar cuando hablemos de funciones o de operaciones.

Algo importantísimo de PHP es que se encarga de realizar las transformaciones necesarias de forma automática.

```
$mivar = 123;
echo $mivar;
```

En el ejemplo anterior, PHP convierte el valor *entero* 123 a la *cadena* de texto "123" antes de pasárselo a la función echo.

También se puede forzar la conversión a un tipo específico, por ejemplo:

```
$mivar = (string)123;
```

Y se puede cambiar el tipo de una variable con:

```
$mivar = 12;
settype($mivar, "double");
```

2.4.1 Ámbito

El ámbito de una variable hace referencia a dónde está disponible esa variable y dónde no, y depende del contexto en el que haya sido definida la variable:

- En el cuerpo de un fichero, las variables son GLOBALES al fichero y a cualquier código que se haya incluido con los comandos "include" o "require" (explicados más adelante.)
- En una función, son LOCALES a esa función y no pueden ser accedidas desde fuera.
- Dentro de una clase, sólo pueden ser accedidas a través del operador "->" sobre el nombre del objeto (las clases se explicarán con detalle más tarde.)

La variable global $mivar es distinta de la variable $mivar que se encuentra dentro de mifuncion().

```
$mivar = 3;
function mifuncion() {
  echo $mivar;
}
```

Para acceder a las variables globales desde una función, hay que utilizar la palabra reservada *global*:

```
$mivar = 3;
function mifuncion() {
  global $mivar;
  echo $mivar;
}
```

2.5 Referencias

En PHP se puede definir "alias" para las variables, es decir, tener dos (o más) nombres distintos para un mismo dato. Se puede ver de forma similar a tener dos punteros en C haciendo referencia a la misma zona de memoria.

Para definir una referencia utilizamos el carácter '&' delante de la variable referenciada:

```
$alias = &$variable
```

Así podremos acceder al mismo dato por *$alias* o *$variable*. Las modificaciones hechas sobre una u otra repercuten en el mismo dato. Debe quedar claro que la referencia no significa que *$alias* "apunta a" *$variable*, si no que tanto *$alias* como *$variable* "apuntan a" un mismo contenido en memoria.

Se puede eliminar una referencia con la función *unset()*:

```
$a = 1;
$b = &$a;
unset ($a); // Pero $b sigue valiendo 1
```

Las referencias también se pueden usar para pasar o devolver parámetros por referencia en las funciones.

2.6 Tipos de datos

PHP soporta los siguientes tipos de datos básicos:

2.6.1 Enteros (*int, integer*)

Números enteros en notación decimal, octal (un 0 inicial indica que el valor está representado en octal) o hexadecimal (un 0x indica que es hexadecimal.)

Ejemplos:

```
$var1 = 1234;    // Número en decimal
$var2 = -1234;   // El mismo número, negativo
$var3 = 0123;    // El 83 decimal, expresado en octal
$var4 = 0x12;    // Valor hexadecimal del 18
```

El tamaño, representación interna y valores máximos dependen de la plataforma, aunque lo normal son 32 bits con signo (+- 2 billones.)

2.6.2 Números en punto flotante (*float, double, real*)

Los números en punto flotante admiten decimales y su rango de valores es mayor que el de los enteros, a costa de cierta pérdida de precisión en los dígitos menos significativos del número. Se permiten las dos notaciones típicas:

```
$var1 = 1.234;   // Número con decimales
$var2 = 1.2e3;   // Notación científica, 1.2 * 10^3
```

Internamente se representan mediante 64 bits en formato IEEE (hasta 1.8e308 con una precisión de 14 dígitos decimales.)

2.6.3 Arrays (*array*)

Los arrays representan vectores unidimensionales o multidimensionales. Se definen y acceden como en C, mediante el nombre de la variable que los contiene e indicando el índice (que comienza en el 0) entre corchetes, en lugar de paréntesis como en ASP. Los elementos de un mismo array pueden ser de tipos distintos, p. Ej.:

```
$MiArray[0] = 1;
$MiArray[1] = "hola!!";
$MiArray[2] = 5;

echo $MiArray[1];
```

Como se ve en el ejemplo, no hace falta definir la dimensión del array antes de usarlo (Dim en ASP), ni, por lo tanto, tampoco redimensionarlo a la hora de añadir más elementos (ReDim.)

Si al definir un array omitimos el índice (pero NO los corchetes, *¡estaríamos sobrescribiendo la variable!*), el elemento se asigna a la siguiente posición del array sin definir. Siguiendo con el ejemplo anterior:

```
$MiArray[] = 8   // $MiArray[3] = 8
```

Los arrays de PHP admiten también cadenas como índices, de forma simultánea a los enteros. Un mismo array puede funcionar a la vez de forma indexada (como un vector) o de forma asociativa (como una tabla hash):

```
$MiArray["nombre"] = "Homer";

echo $MiArray[3];                    // 8
echo $MiArray["nombre"];             // Homer
```

Para definir un array multidimensional, simplemente indicamos más índices:

```
$MiOtroArray[1][2]["pepe"][0] = "4 dimensiones!!!";
```

También podemos definir arrays utilizando los constructores del lenguaje *array()* o *list()*, así:

```
$OtroArrayMas = array( 1, "hola", 5);
```

Donde los valores se asignan por orden a los índices 0, 1 y 2, o de esta otra forma, donde indicamos explícitamente el índice:

```
$YOtroArray = array(
   0 => 1,
   1 => "hola",
   2 => 5,
```

```
    3 => 8,
    "nombre" => "Homer"
);
```

Se pueden definir arrays multidimensionales mediante composición de varias llamadas anidadas al constructor *array()*.

2.6.4 Cadenas de texto (*string*)

Las cadenas en PHP se pueden definir de tres formas:

- Si se delimitan entre comillas dobles ("), se expandirá cualquier variable que haya dentro de la cadena. Además, se pueden incluir ciertas secuencias de escape, al igual que en C:

Secuencia	Significado
\n	Nueva línea (LF ó 0x0A en ASCII)
\r	Retorno de carro (CR ó 0x0D en ASCII)
\t	Tabulación horizontal (HT ó 0x09 en ASCII)
\\	Barra invertida
\$	Símbolo del dólar
\"	Dobles comillas
\[0-7]{1,3}	Un carácter determinado en notación octal
\x[0-9A-Fa-f]{1,2}	El carácter indicado en hexadecimal

- Si se delimitan entre comillas simples ('), las variables no se expanden y además las únicas secuencias de escape que se reconocen son "\\" y "\'" (barra invertida y comillas simples.) Debido a estas limitaciones, este segundo método es bastante más rápido que el primero a la hora de manipular o imprimir cadenas, ya que el análisis y proceso al que son sometidas las es menor.

- Utilizando la sintaxis "here doc" de Perl, cuya estructura es:

```
$cadena = <<<DELIMITADOR
texto
texto
texto
...
texto
DELIMITADOR
```

Después del operador "<<<" especificamos un delimitador que marcará el final del texto. Debemos llevar cuidado al elegir este delimitador, ya si aparece en algún lugar del texto, podríamos acabar con un resultado incorrecto. Tras la línea con el "<<<" y el delimitador, escribimos la cadena de texto, que puede expandirse por tantas

líneas como queramos. Las variables dentro del texto se expanden y no hace falta escapar las comillas. Para finalizar la cadena, debe aparecer una línea que contenga únicamente el delimitador. Por ejemplo:

```
$cadena = <<<FINCAD
Esto es un ejemplo de cadena como "here doc".
La variable \$a vale $a.
Ahora vamos a finalizar la cadena:
FINCAD
```

Para concatenar cadenas se utiliza el operador '.' :

```
$cad = 'A esta cadena ';
$cad = $cad . 'le vamos a añadir más texto.';
```

Se puede acceder a cada carácter de la cadena de forma independiente utilizando notación de arrays indexados sobre la cadena:

```
$cad2 = "El tercer carácter de \$cad es '$cad[2]'.";
```

Por último, señalar que aunque como ya hemos dicho en una cadena delimitada por dobles comillas se expanden las variables, con construcciones complejas (arrays multidimensionales, objetos...) la expansión NO siempre funciona bien. Para evitar problemas, podemos concatenar la cadena con el valor de la variable o encerrar la variable entre llaves:

```
echo "Esto no irá bien $a[1][3]";
echo "Así no hay problemas {$a[1][3]}";
echo 'Concatenar es otra alternativa ' . $cosa->valor;
```

2.6.5Mayúsculas y minúsculas

Esta puede ser una causa de problemas. En ASP no importa si una función o variable está escrita en mayúsculas o minúsculas, siempre se interpretará igual. En cambio en PHP tenemos un comportamiento mixto:

- En las variables, las mayúsculas y minúsculas IMPORTAN. Así, la variable $MiVar es distinta de $mivar.
- En los nombres de funciones y palabras reservadas, las mayúsculas NO IMPORTAN. La función PRINT() hace referencia a print().

Para evitar errores y confusiones, siempre escribiremos los nombres de funciones del sistema en minúscula, y las funciones propias siempre tal y como se escribieran en la declaración.

2.6.6 Objetos (*object*)

Pese a no ser un Lenguaje Orientado a Objetos *puro*, PHP soporta clases y objetos, aunque con ciertas carencias respecto a otros lenguajes. Las clases y objetos en PHP se definen y usan de forma similar a C++ o Java.

2.7 Constantes

En PHP podemos definir constantes utilizando la función *define()*, cuya declaración es:

```
int define(string nombre, mixed valor [, int noMayusculas])
```

Donde *nombre* es el nombre que le queremos dar a la constante, *valor* su valor, y el campo opcional *noMayusculas* indica si está a 1 que podemos acceder a la variable independientemente con mayúsculas o minúsculas, mientras que si está a 0 (valor por defecto) sólo podremos acceder a ella de la misma forma como la hayamos definido.

Las constantes en PHP se diferencian de las variables en que:

- no llevan el símbolo del dólar delante.
- puede accederse a ellas desde cualquier parte del código donde han sido definidas, sin restricciones de ámbito como en las variables.
- no pueden ser redefinidas o borradas una vez definidas.
- sólo pueden contener valores escalares, no vectores.

Un ejemplo de declaración y uso de constantes en PHP sería:

```
define('SALUDO', 'Hola, mundo!');
echo 'La constante SALUDO vale ' . SALUDO;
```

Operadores

Vamos a ver los distintos operadores disponibles en PHP, clasificados por tipos:

2.8 Operadores aritméticos

Disponemos de los clásicos operadores aritméticos:

Operación	Nombre	Resultado
$a + $b	Suma	Suma de $a y $b.
$a - $b	Resta	Diferencia entre $a y $b.
$a * $b	Multiplicación	Producto de $a y $b.
$a / $b	División	Cociente de $a y $b.
$a % $b	Módulo	Resto de la operación $a / $b.

2.8.1 Auto-incremento y auto-decremento

PHP también dispone de los típicos operadores de auto-incremento y decremento de C:

Operación	Nombre	Resultado

Operación	Nombre	Resultado
++$a	Pre-incremento	Incrementa $a en 1, y devuelve $a (ya incrementado)
$a++	Post-incremento	Devuelve $a (sin incrementar), y después lo incrementa en 1.
--$a	Pre-decremento	Decrementa $a en 1, y después lo devuelve.
$a--	Post-decremento	Devuelve $a, y después lo incrementa en 1.

Por si alguien no está familiarizado con el funcionamiento de estos operadores, he aquí un ejemplo para clarificarlo:

```
$a = 1;
$b = $a++;        // $b = $a; $a = $a + 1; -> $b vale 1, $a 2.

$a = 1;
$b = ++$a;        // $a = $a + 1; $b = $a; -> $a vale 2, y $b 2.
```

2.8.3 Operadores de bits

Veamos ahora los operadores BIT a BIT de que dispone PHP:

Operación	Nombre	Resultado
$a & $b	Y	Se ponen a 1 los bits que están a 1 en $a y $b.
$a | $b	O	Se ponen a 1 los bits que están a 1 en $a o $b.
$a ^ $b	O Exclusivo	Se ponen a 1 los bits que están a 1 en $a o $b, pero no en ambos.
~ $a	No	Se invierten los bits (se cambian 1 por 0 y viceversa.)
$a << $b	Desp. Izq.	Desplaza $b posiciones a la izquierda todos los bits de $a.
$a >> $b	Desp. Drch.	Desplaza $b posiciones a la derecha todos los bits de $a.

2.8.4 Operadores lógicos

Los operadores lógicos realizan operaciones dependiendo del valor booleano de los operandos.

Operación	Nombre	Resultado
$a and $b	Y	Cierto si $a y $b son ciertos.
$a or $b	O	Cierto si $a o $b es cierto.
$a xor $b	O Exclusivo.	Cierto si $a o $b es cierto, pero no ambos.
! $a	No	Cierto si $a es falso.
$a && $b	Y	Cierto si $a y $b son ciertos.
$a || $b	O	Cierto si $a o $b es cierto.

La razón de que haya dos operadores distintos para las operaciones *Y* y *O* lógicas es que tienen *distinta precedencia* (ver punto 2.8.9.)

2.8.5 Asignación, igualdad e identidad

En PHP hay tres operadores distintos para asignar y comparar valores entre variables, mientras que en ASP todo se realiza con el único operador "=":

Operación	Nombre	Resultado
$a = $b	Asignación	Asigna el valor de una variable o expresión del segundo término a la variable del primer término.
$a == $b	Igualdad	Compara si el valor de los dos operandos es el mismo.
$a === $b	Identidad	Compara si el valor es el mismo y, además, el tipo coincide.

Ejemplo:

```
$var1 = 1;                  // Asignación
$var2 = 1;
$var3 = "1";
($var1 == $var2)            // Cierto, son iguales
($var1 == $var3)            // Son iguales (tras la conversión)
($var1 === $var2)           // Cierto, son idénticas
($var1 === $var3)           // FALSO, el tipo no coincide
```

Los programadores acostumbrados a ASP deben llevar mucho cuidado con esto, ya que puede llevar a errores de este tipo:

```
$var1 = 1;
$var2 = 2;
if( $var1 = $var2 ) {
   echo 'iguales';
} else {
   echo 'distintas';
}
```

Esta condición en ASP se evaluaría a FALSO. En cambio, en PHP como '=' es el operador de ASIGNACIÓN y no el de IGUALDAD, lo que estamos haciendo es asignar el valor de *$var2* a *$var1*. Tras esto *$var1* vale '2', que como es distinto de 1 se evaluará a CIERTO.

2.8.6 Comparaciones

Devuelven cierto o falso según el resultado de comparar los dos operandos.

Operación	Nombre	Resultado
$a != $b	No igual	Cierto si el valor de $a no es igual al de $b.

Operación	Nombre	Resultado
$a !== $b	No idéntico	Cierto si $a no es igual a $b, o si no tienen el mismo tipo.
$a < $b	Menor que	Cierto si $a es estrictamente menor que $b.
$a > $b	Mayor que	Cierto si $a es estrictamente mayor que $b.
$a <= $b	Menor o igual que	Cierto si $a es menor o igual que $b.
$a >= $b	Mayor o igual que	Cierto si $a es mayor o igual que $b.

La principal diferencia con ASP aquí es que para expresar la desigualdad, en PHP se utilizan "!=" y "!==" mientras que en ASP se utiliza "<>".

También se puede englobar aquí el operador condicional "?:", que funciona como en C y otros lenguajes:

```
(expr1) ? (expr2) : (expr3);
```

Esta expresión devuelve *expr2* si *expr1* se evalúa a cierto, o *expr3* si *expr1* se evalúa a falso. Por ejemplo:

```
$cad = $a > $b ? "a es mayor que b" : "a no es mayor que b";
```

2.8.7 Operadores de cadenas de texto.

Para operar con cadenas sólo disponemos de un operador: la concatenación de cadenas representada por el punto '.'.

Ej. :

```
$a = 1;
$b = 2;
$c = 'El resultado de ' . $a . ' + ' . $b . ' es ' . $a + b;
```

Que dejaría en *$c* la cadena "El resultado de 1 + 2 es 3". Antes de cada concatenación se realizarán las conversiones de tipo que fueran necesarias (en el ejemplo, los enteros se convierten a cadenas.)

2.8.8 Atajos en la asignación

Al igual que en C, C++ y Java, en PHP disponemos de una serie de "atajos" para, en una sola operación, operar sobre una variable y asignarle a esa misma variable el resultado.

Las operaciones susceptibles de ser usadas en estos atajos son:

```
+ - * / % & ^ . >> y <<
```

resultando en los nuevos signos de operación-asignación:

+= -= *= /= %= &= ^= .= >>= y <<=

Ejemplos de uso:

```
$var1 += 3;      // $var1 = $var1 + 3;
$var2 /= 2;      // $var2 = $var2 / 2;
$var3 >>= 1;     // $var3 = $var3 >> 1;
```

2.8.9 Precedencia y asociatividad de operandos

La precedencia de los operandos resuelve el orden en el que se evalúa una expresión múltiple que no ha sido delimitada con paréntesis. Por ejemplo, *1 + 5 * 3* en PHP daría como resultado *1 + (5 * 3) = 16* y no *(1 + 5) * 3 = 18* ya que el producto tiene mayor precedencia que la suma.

La tabla muestra la asociatividad de los operandos en PHP, y está ordenada en orden decreciente de precedencia (los más prioritarios primero):

Asociatividad	Operandos
izquierda	,
izquierda	or
izquierda	xor
izquierda	and
derecha	print
izquierda	= += -= *= /= .= %= &= |= ^= ~= <<= >>=
izquierda	? :
izquierda	||
izquierda	&&
izquierda	|
izquierda	^
izquierda	&
no-asociativo	== != === !==
no-asociativo	< <= > >=
izquierda	<< >>
izquierda	+ - .
izquierda	* / %
derecha	! ~ ++ -- (int) (double) (string) (array) (object) @
derecha	[
no-asociativo	new

2.9 Estructuras de control

Las estructuras de control permiten bifurcar el flujo del programa y así ejecutar unas partes u otras del código según ciertas condiciones. PHP dispone de todas las estructuras clásicas de los lenguajes de alto nivel, con la sintaxis de C, C++ o Java, y además algunas otras estructuras más típicas de lenguajes interpretados como Perl o Bash.

Las estructuras de control contienen una *expresión* cuya evaluación a cierto o falso determinará el flujo a seguir dentro de la estructura. Estas expresiones pueden ser una variable, una función (el valor que devuelve), una constante, o cualquier combinación de éstas con los operadores vistos en el punto anterior.

2.9.1 Bifurcaciones

2.9.1.1 if... elseif... else

El condicional *if* es la estructura de control más básica de todas. En su forma más simple, su sintaxis es esta:

```
if (expresión) {
   comandos
}
resto
```

Su funcionamiento es idéntico al de ASP, salvando las diferencias sintácticas: Si *expresión* se evalúa a cierto, se ejecutan los *comandos* y después se sigue ejecutando el *resto* del programa. Si se evalúa a falso, no se ejecutan los *comandos* y continúa con el *resto*. Las únicas diferencias con ASP es que *expresión* debe ir entre paréntesis SIEMPRE, que no hay que poner la palabra *THEN* y que el cuerpo se delimita con llaves en lugar de con *IF .. ENDIF*. Estas diferencias de PHP frente a ASP (expresión entre paréntesis y delimitación del cuerpo con llaves) van a ser las mismas para todas las clases de estructuras de control.

Las llaves '{' y '}' no son necesarias (ni en el if ni en ninguna otra sentencia de control) si sólo hay un comando que ejecutar tras la condición, sólo hay que ponerlas para agrupar si hay más de un comando. A pesar de este carácter optativo es aconsejable acostumbrarse a ponerlas siempre, aunque sólo haya un comando que ejecutar, ya que si al continuar el desarrollo se decide añadir un segundo comando (o varios) tras la condición es fácil olvidarse de añadir las llaves y el resultado sería incorrecto.

A una sentencia *if* le podemos añadir código que se ejecute cuando la condición no se verifica mediante la sentencia *else*:

```
if (expresión) {
   comandos_cierto
} else {
   comandos_falso
}
```

Si *expresión* se evalúa a cierto, se ejecutan *comandos_cierto*. Si se evalúa a falso, se ejecuta *comandos_falso*. En ambos casos luego se ejecuta el resto de comandos que sigan a la instrucción *if*.

Por último, podemos encadenar varias condiciones con la sentencia *elseif*, de esta forma:

```
if (expresion1) {
   comandos1
} elseif (expresion2) {
   comandos2
} elseif (expresion3) {
   comandos3
}
...
elseif (expresionN) {
   comandosN
} else {
   comandosElse
}
```

El flujo del código comienza evaluando *expresión1*. Si es cierta, ejecuta *comandos1*. Si no, evalúa *expresión2*. Si es cierta, ejecuta *expresión2*. Si no, evalúa *expresión3*... y continúa así hasta que alguna de las condiciones de un *endif* se verifique. Si no se verifica ninguna, se ejecuta *comandosElse* (este último *else* es optativo.) En cualquier caso, después se continúa con el flujo normal del programa.

2.9.1.2 S*witch*

La estructura *switch* de PHP es equivalente al *SELECT CASE* de ASP. Su sintaxis es:

```
switch (variable) {
   case valor1:
         comandos1
   case valor2:
      comandos2
   ...
   case valorN:
      comandosN
   default:
      comandosDefault
}
```

El flujo procede linealmente de arriba a abajo, comparando con cada valor de los case, y ejecutando el código asociado si se cumple la condición. En caso de que no se cumpla ninguna, se ejecuta el código asociado a la cláusula *default*. El comportamiento sería similar a tener un *if* por cada *case*, uno detrás del otro.

Las estructuras de control *while* y *do .. while* representan bucles que se ejecutan mientras se verifique una determinada condición, igual que *DO .. LOOP* y *WHILE .. WEND* en ASP. La estructura de un bucle *while* es esta:

```
while (expresión) {
   comandos
}
```

Y el resultado es que se estará ejecutando toda la serie de sentencias especificada en *comandos* mientras que *expresión* se evalúe a cierto.

Llegados a este punto hay que hacer notar dos detalles:

- Si la primera vez que el flujo del programa llega a la sentencia *while*, *condición* se evalúa a falso, *comandos* no se ejecuta.
- Si *expresión* se evalúa a cierto y dentro de *comandos* no se modifica el valor de alguna de las variables contenidas en *expresión*, el código entrará en un bucle infinito (a no ser que algún proceso externo sea el encargado de variar el valor de *expresión*.)

La estructura *do .. while* puede ser vista como una variante de *while* en la que la comprobación de la condición se realiza al final de cada iteración del bucle en lugar de al principio. Con esto lo que se consigue es que al menos la iteración se realice siempre una vez, aunque expresión se evalúe a falso. La estructura de *do .. while* es esta:

```
do {
   comandos
} while (expresión);
```

2.9.2.3 break

La sentencia *break* nos permite salir *inmediatamente* de una estructura de control *while*, *for* o *switch*. Por ejemplo, el siguiente bucle *while* finalizaría en la iteración *$a = 5*, pese a que la condición del *while* es *$a < 10*:

```
$a = 0;
while ($a < 10) {
   if ($a == 5) {
      break;
   }
   $a++;
}
```

Después del *break* podemos especificar un parámetro, el número de niveles de bucles anidados de los que queremos salir. Por defecto es uno (salir del bucle más interno) :

```
$a = 0;
```

Hay que destacar como diferencia respecto a ASP que cuando se termina de ejecutar el código de un *case*, si no se finaliza el *switch* explícitamente con un *break*, se continúa ejecutando el código del siguiente case aunque no se cumpla la condición, hasta que se llegue al final del bloque *switch* o se finalice este con un *break*. Por ejemplo:

```
switch ($i) {
   case 1:
      echo "Código del 1";

   case 2:
      echo "Código del 2";

   case 3:
      echo "Código del 3";
      break;

   case 4:
      echo "Código del 4";
}
```

Si *$i* vale 1, se imprimirán las tres primeras cadenas; Si vale 2, la segunda y la tercera; Si vale 3, sólo la tercera; Y si vale 4, sólo la última.

Otro ejemplo del funcionamiento del *switch*, ahora con una cláusula *default*:

```
switch ($i) {
   case 0:
   case 1:
   case 2:
   case 3:
      echo "i es menor que 4, pero no negativa";
      break;

   case 4:
      echo "i vale 4";
      break;

   default:
      echo "i mayor que 4 o negativa";
      break;
}
```

2.9.2 Bucles

2.9.2.1 y 2.9.2.2 while y *do .. while*

```
while ($a < 10) {
   $b = 0;

   while ($b < 5) {
      if ($b == 2) {
         break; // Equivale a "break 1", sale del while b
      }
   }

   while ($b < 5) {
      if ($a == 3 && $b == 3) {
         break 2; // Saldría de los DOS bucles.
      }
   }
}
```

2.9.2.4 Continue

Por su parte, la sentencia *continue* lo que hace es saltarse el resto de la iteración actual, y pasar directamente a la siguiente:

```
$a = 0;
while ($a < 5) {
   if ($a == 2) {
      continue;
   }
   echo "\$a vale $a.";
}
```

En el ejemplo se saltaría el *echo* de la iteración *$a = 2*, y el resultado sería:

```
$a vale 0
$a vale 1
$a vale 3
$a vale 4
```

2.9.2.5 For

Los bucles *for* son los más complejos de que dispone PHP, bastante más que sus homónimos de ASP. Su estructura es la misma que en C:

```
for (expresión1; expresión2; expresión3) {
  comandos
}
```

donde:

- *expresión1* es la iniciación del bucle. Generalmente da un valor inicial a una o varias variables (separadas por comas). Sólo se ejecuta una vez, al principio, cuando el flujo del programa llega al bucle.
- *expresión2* es la condición. Mientras que *expresión2* se evalúe a cierto, el bucle estará iterando. Se evalúa al inicio de cada iteración, y si no se verifica la condición la siguiente iteración ya no se realiza y finaliza el bucle, continuando la ejecución del programa con el resto del código de después del *for*.
- *expresión3* es el paso de iteración. Se ejecuta después de cada iteración, y generalmente modifica el valor de alguna variable (separadas por comas si hay más de una).

Cualquiera de las tres expresiones puede estar vacía, aunque en este caso tendremos que llevar cuidado de realizar su función en el cuerpo del bucle.

Ejemplos:

```
$factorial5 = 1;
for ($i = 2; $i <= 5; $i++ ) {
   $factorial5 *= $i;
}
```

El bucle anterior calcula la factorial de 5 (5!). En *expresión1* podemos inicializar varias variables separándolas con comas, con lo que el código se podría rescribir así:

```
for ($factorial5 = 1, $i = 2; $i <= 5; $i++ ) {
   $factorial5 *= $i;
}
```

Por último, en *expresión3* también podemos operar sobre varias variables separándolas con comas, con lo que podríamos encerrar todo el código del bucle en la línea *for* de esta forma:

```
for ($factorial5=1, $i=2; $i<=5; $factorial5*=$i, $i++);
```

En general no se debe complicar tanto un bucle *for* porque como se ve se pierde bastante en la claridad del código. Esto es un ejemplo tanto de la potencia de los bucles *for*, como de un mal uso (abuso) de ellos.

Los cuatro ejemplos siguientes tienen el mismo resultado: muestran los números del 0 al 10.

```
/* ejemplo 1 */

for ($i = 1; $i <= 10; $i++) {
   print $i;
}

/* ejemplo 2 */

for ($i = 1;;$i++) {
```

```
   if ($i > 10) {
      break;
   }
   print $i;
}

/* ejemplo 3 */

$i = 1;
for (;;) {
   if ($i > 10) {
      break;
   }
   print $i;
   $i++;
}

/* ejemplo 4 */

for ($i = 1; $i <= 10; print $i, $i++) ;
```

De nuevo, algunos de estos ejemplos (en especial el último) no lo son de buenas costumbres de programación, pero sí de la potencia y flexibilidad del bucle *for*.

Como hemos visto, el bucle *for* de PHP es MUCHO más potente que el de ASP, así que no debería haber problemas a la hora de pasar bucles de este tipo de ASP a PHP. Valga como ejemplo:

```
' ASP
<%FOR i=1 TO 100%>
<%=MiVar%>
<%NEXT%>

// PHP
<?php
for ($i = 1; $i <= 100; $i++) {
   echo $MiVar;
}
?>
```

2.9.2.6 Foreach

El bucle *foreach* es nuevo en PHP4, y representa una estructura de control típica de lenguajes interpretados como Perl y Bash, en la que a una variable se le van asignando todos los valores de una lista. Su sintaxis es esta:

```
foreach (array as $variable) {
   comandos
```

```
}
```

Es equivalente al *FOR EACH .. NEXT* de ASP. En cada iteración del bucle, se coloca en $*variable* un elemento de *array*, comenzando por el primero y siguiendo un orden ascendente. Por ejemplo:

```
$a = array (1, 2, 3, 17);

foreach ($a as $v) {
   print "Valor actual de \$a: $v.\n";
}
```

El resultado sería:

```
Valor actual de $a: 1
Valor actual de $a: 2
Valor actual de $a: 3
Valor actual de $a: 17
```

2.10 Evaluaciones a Cierto o Falso

La forma de realizar las comprobaciones booleanas sobre una variable (evaluar su valor a cierto o falso) en PHP puede resultar confusa a los programadores acostumbrados a trabajar con lenguajes fuertemente tipados, ya que antes de realizar la comprobación PHP puede convertir el tipo de la variable. En PHP se puede evaluar de esta forma cualquier variable, contenga un valor del tipo que contenga, lo que lleva a varios tipos de conversión:

Para los valores numéricos, 0 es FALSO, cualquier otro valor CIERTO.

```
$x = 1; // $x
if( $x ) // se evalúa a cierto
$x = 0; // $x definida como el entero 0
if( $x ) // se evalúa a falso
```

Para cadenas de texto, una cadena vacía equivale a FALSO, una cadena no vacía a CIERTO.

```
$x = "hello"; // asignamos una cadena a $x
if( $x )     // se evalúa a cierto
$x = "";     // cadena vacía
if( $x )     // evalúa a falso
```

NOTA: $x = "0" es la única excepción, ya que primero se convierte la cadena "0" al decimal 0, que como ya hemos visto se evalúa a FALSO.

Para arrays: un array vacío se evalúa a FALSO, mientras que si tiene algún elemento lo hace a CIERTO.

```
$x = array(); // $x es un array vacío
if( $x )      // se evalúa como falso
$x = array( "a", "b", "c" );
if( $x )      // se evalúa a cierto
```

Para objetos, el resultado de la evaluación es FALSO si son objetos vacíos (su clase no define ningún método ni variable), y CIERTO en otro caso.

```
Class Yod {}  // clase vacía
$x = new Yod();
if( $x )      // se evalúa a falso
Class Yod {   // clase no vacía
   var $x = 1;
}
$x = new Yod();
if( $x )      // se evalúa a cierto
```

PHP tiene definidas dos constantes para los valores CIERTO y FALSO, respectivamente TRUE y FALSE. Están definidas de esta forma:

- TRUE es el valor entero decimal 1.
- FALSE es la cadena vacía.

Es indiferente si se escriben en mayúsculas o minúsculas, es decir, 'true', 'True' y 'tRuE' hacen referencia a la misma constante 'TRUE'. Para evitar confusiones, escribiremos estas constantes siempre en mayúsculas.

Como regla general a utilizar al principio hasta que nos acostumbremos al funcionamiento de PHP, podemos tener en mente esta regla que funciona el 99% de las veces: el valor 0 y la cadena vacía se evalúan a FALSO, cualquier otra cosa a CIERTO.

2.11 Funciones

Las funciones de PHP serían el equivalente a los procedimientos *SUB* y *FUNCTION* de ASP. Con una función podemos agrupar bajo un nombre una serie de comandos que se repiten a menudo a lo largo del código, y en vez de repetir este código varias veces lo sustituimos por una simple "llamada" a la función.

2.11.1 Paso de parámetros por valor

La sintaxis de la declaración de una función en PHP es:

```
function nombre ($arg_1, $arg_2, ..., $arg_n) {
   comandos
   return $salida;
}
```

Los parámetros se pasan por valor, es decir, se crea en *$arg_1 ... $arg_n* copias locales de las variables, y se trabaja sobre estas copias locales, de forma que al salir de la función los valores originales no han sido modificados. El comando *return* es opcional, y sirve para que la función devuelva un valor de salida. Es el equivalente a asignar un valor al nombre de la función en ASP. Puede aparecer varias veces en el código de la función, y siempre implica el final de la ejecución de la función. En el cuerpo de la función puede haber cualquier combinación de instrucciones válidas en PHP, incluso otras definiciones de funciones y / o clases. Por ejemplo:

```
function factorial ($valor) {
   if ($valor < 0) {
      return –1; // Error
   }

   if ($valor == 0 ) {
      return 1;
   }

   if ($valor == 1 || $valor == 2) {
      return $valor;
   }

   $ret = 1;
   for ($i = 2; $i <= $valor; $i++) {
      $ret *= $i;
   }
   return $ret;
}

$factorial5 = factorial(5);
```

En PHP 3 era necesario definir las funciones antes de usarlas, como en el ejemplo de arriba. Esta restricción se ha eliminado en PHP 4, donde ya se puede utilizar una función antes de su declaración.

PHP no soporta sobrecarga de funciones (tener varias funciones con el mismo nombre y distintos argumentos), ni tampoco se puede eliminar o modificar una función previamente definida. Lo que si se puede hacer es dar valores por defecto a algunos de los parámetros que reciba la función (comenzando siempre por la derecha) y hacerlos así optativos:

```
function enlace($url = "www.php.net") {
   echo '<a href="" . $url . "">Pulsa aquí</a>';
}
```

2.11.2 Paso de parámetros por referencia

Los parámetros de las funciones se pueden pasar por referencia, de forma que si que se pueda modificar su valor dentro de la función. Esto se consigue utilizando el símbolo '&' en la definición de la función, que tiene el mismo efecto que *ByRef* en ASP:

```
function MiFuncion(&$var) {
   $var++;
}

$a = 5;
MiFuncion($a);
// Aquí $a == 6
```

2.11.3 Devolución de variables por referencia

PHP también nos permite devolver variables por referencia. Esto puede ser útil, por ejemplo, cuando tenemos una función que busca un valor dentro de una colección de variables, y queremos que devuelva la variable entera (porque es una estructura que contiene más datos que vamos a necesitar.) En este caso, hay que utilizar el '&' tanto en la definición de la función como en la llamada a ésta:

```
function &buscar_cliente($nombre) {
   // ... buscamos ...
   return $registro;
}

$cliente = &buscar_cliente("Juan");
echo $cliente->dni;
```

2.11.4 Funciones para la manipulación de strings y arrays

2.11.4.1 Comparaciones

Para comparar cadenas ya hemos visto que podemos utilizar los operadores == y ===, aunque también disponemos de la función *strcmp()* con el mismo funcionamiento que en C:

```
int strcmp (string str1, string str2)
```

La función devuelve 0 si ambas cadenas son iguales, un número menor que cero si *$a* es menor que *$b*, y mayor que cero si *$a* es mayor que *$b*. Además, siempre que la cadena contenga algún carácter binario deberemos utilizar esta función en lugar del operador "==".

Por ejemplo:

```
if (strcmp($a, $b) == 0) {
   echo 'iguales';
```

```
}
```

La función *strcmp* tiene en cuenta mayúsculas y minúsculas. Si queremos comparar sin tenerlas en cuenta, usaremos *strcasecmp*:

```
int strcasecmp (string str1, string str2)
```

2.11.4.2 Subcadenas

Otra operación que podríamos necesitar es obtener una subcadena de otra dada. Esto se consigue con:

```
string substr (string cadena, int inicio [, int tamaño])
```

Si *inicio* es positivo, devuelve la subcadena que empieza en esa posición. Si es negativo, la subcadena que empieza en esa posición contando desde el final. Si se indica el *tamaño*, se devuelve una subcadena de hasta ese número de caracteres. Si el *tamaño* es negativo, se elimina ese número de caracteres de la subcadena devuelta:

```
$str = substr('abcdef', 2, 3);   // cde
$str = substr('abcdef', -2);     // ef
$str = substr('abcdef', -2, 1);  // e
$str = substr('abcdef', 1, -2);  // bcd
```

En ocasiones puede que no sepamos la posición exacta en la cadena de lo que andamos buscando, pero si una serie de caracteres de *referencia* que marcarán su inicio. Entonces podremos usar estas funciones:

```
int strpos (string cadena, string referencia [, int inicio])
int strrpos (string cadena, char referencia)
string strstr (string cadena, string referencia)
```

La primera función devuelve el índice de la primera ocurrencia de la cadena *referencia* en *cadena* a partir de la posición *inicio*. La segunda función es similar pero aquí *referencia* es un carácter en lugar de una cadena (si se pasa una cadena sólo se usará el primer carácter), y se busca desde el final de la cadena. Por último, la tercera función devuelve la subcadena de *cadena* que comienza en la primera ocurrencia de *referencia*.

```
$i = strpos('cadena de prueba', 'de');
// $i = 2

$i = strpos('cadena de prueba', 'de', 5);
// $i = 7

$s = strrpos('cadena de prueba', 'de');
```

```
// $i = 7

$s = strstr('cadena de prueba', 'de');
// $s = dena de prueba
```

2.11.4.3 Imprimir y formatear cadenas

Se puede formatear e imprimir una cadena con la función *printf()*, al igual que en C:

```
int printf (string formato [, mixed args...])
```

Aquí, formato es una cadena que puede contener cualquier carácter excepto '%', que se imprimirán tal cual, y secuencias de formato que comienzan por '%' y controlan cómo se mostrarán los argumentos. Estas secuencias de formato se componen de:

1. Un carácter opcional que se utilizará para rellenar y ajustar el tamaño del campo. Por ejemplo, un espacio para caracteres o un cero para números. Por defecto se usa el espacio.
2. Un indicador opcional para especificar si se alineará a la izquierda (-). Por defecto se alinea a la derecha.
3. El número mínimo de caracteres que ocupará el campo tras la conversión. También es opcional.
4. Un indicador opcional de precisión, formado por un punto seguido del número de dígitos decimales que se deberán mostrar con los números en punto flotante. No tiene efecto con otros tipos de datos.
5. Un indicador de tipo que especifica cómo se deberá tratar el dato. Los tipos disponibles son:

%	El carácter de tanto por ciento.
b	El argumento se trata como entero y se muestra en binario.
c	Se trata como entero, y se muestra el carácter ASCII.
d	Se trata como entero y se muestra el número en decimal.
f	Se trata como double, y se muestra como punto flotante.
o	Se trata como entero y se muestra en octal.
s	El argumento se trata y se muestra como una cadena.
x	Se trata como entero y se muestra en hexadecimal (con las letras en minúsculas).
X	Se trata como entero y se muestra en hexadecimal (con las letras en mayúsculas).

Ejemplos:

```
printf("%02d/%02d/%04d", $dia, $mes, $año);

$pago1 = 68.75;
$pago2 = 54.35;
$pago = $pago1 + $pago2;
// echo $pago mostrará "123.1"
```

```
// Mostrar al menos un dígito entero y exactamente dos
// decimales, rellenando con ceros
printf ("%01.2f", $pago);
```

También podemos almacenar el resultado del formateo en una cadena en lugar de imprimirlo con la función *sprintf()*:

```
string sprintf (string formato [, mixed args...])
```

El formato es el mismo que con *printf*:

```
$fecha = sprintf("%02d/%02d/%04d", $dia, $mes, $año);
```

Estas dos funciones son muy útiles a la hora de dar un formato específico a unos datos, pero se debe de huir de ellas cuando este formato no sea importante o simplemente no se esté realizando ningún formateo, ya que es mucho más rápido imprimir con *echo* o concatenar cadenas con el operador '.'.

2.11.4.4 Escapar caracteres

PHP tiene varias funciones para "escapar" con barras invertidas algunos caracteres que bajo ciertas circunstancias pueden dar problemas. Por ejemplo, en SQL tendremos que escapar los apóstrofes en las restricciones de un WHERE, pero si por ejemplo estas restricciones las hemos obtenido por parte del usuario (en un form), no sabemos a priori si habrá algún carácter que escapar ni dónde. Con la función *addslashes()* podemos manejar este tipo de situaciones, ya que se encarga de añadir las barras invertidas que haga falta:

```
$busca = "D'Alton"; // Habrá que escapar el apóstrofe
$sql = 'SELECT * FROM usuarios WHERE apellido = \" .
      addslashes($busca) . '\";
```

Otro caso en el que escapar caracteres es MUY importante es a la hora de realizar llamadas al sistema. En PHP podemos ejecutar cualquier comando en el servidor con la función *system()*:

```
string system (string comando [, int valor_salida])
```

Por ejemplo, podríamos tener una página en la que en un *form* se pida el nombre de un usuario de nuestro sistema, y que devuelva la información de *finger* sobre ese usuario. Si lo hiciéramos simplemente así:

```
echo system("finger $usuario");
```

y en *$usuario* tenemos la entrada del form sin más, tendríamos un grave agujero de seguridad, ya que si en el form por ejemplo nos hubieran puesto "pepe ; apachectl stop" y PHP se estuviera ejecutando como root cualquiera nos podría detener Apache. Para evitar esto habría que aplicar esta función sobre los datos que recibamos del form:

```
string escapeshellcmd (string comando)
```

que se encarga de escapar con barras los caracteres que se podrían usar en el shell de UNIX para ejecutar un programa sin nuestro permiso (en concreto, #&;`'\"|*?~<>^()[]{}$\\\x0A\xFF).

Además de por razones de seguridad, también necesitaremos a veces cambiar unos caracteres por otros para formatear correctamente un texto en HTML. Esto lo hace la función *htmlspecialchars()*:

```
$valor = "a>b";
echo '<input type=hidden name=var value="' .
    htmlspecialchars($valor) . '">';
// <input type=hidden name=var value="a&gt;b">
```

Podemos también convertir todos los caracteres de fin de línea de una cadena a "
" con:

```
string nl2br (string cadena)
```

2.11.4.5 Extraer campos

Podemos dividir una cadena formateada en campos divididos por un delimitador en un array de cadenas con esos campos usando la función *explode()*:

```
array explode (string delimitador, string cadena
        [, int límite])
```

Si se especifica un *límite*, se extraerá hasta ese número de campos y en el último estará el resto de la cadena.

Por ejemplo, en UNIX se utiliza bastante el carácter de dos puntos para separar campos en una cadena, por ejemplo en el fichero /etc/passwd. Podemos utilizar esta función para obtener el valor de cada campo.

```
$cadena = "campo1:campo2:campo3";
$campos = explode(":", $cadena);
```

La operación inversa, por la que a partir de un array de campos y un separador se junta todo en una sola cadena, es *implode()*:

```
string implode (string delimitador, array campos)
```

Para volver a la cadena original del ejemplo anterior, usaríamos:

```
$cadena = implode(":", $campos);
```

En caso de que delimitador no siempre sea el mismo carácter, tendremos que recurrir a *split*, cuyo funcionamiento es similar al de *explode* pero utiliza expresiones regulares para dividir los campos:

```
array split (string delimitador, string cadena
        [, int límite])
```

Por ejemplo, para obtener los campos de una fecha donde el día, mes y año puedan estar separados por espacios, barras, guiones o puntos, tendríamos que utilizar *split* así:

```
$fecha = "12/4 2000";
$campos = split ('[ /.-]', $fecha);
```

2.11.4.6 Recorrer un array

Todos los arrays de PHP disponen de un puntero que señala al elemento *actual*, puntero que sirve para recorrer el array secuencialmente con las funciones *current(), next(), prev(), reset()* y *end()*, que devuelven el elemento actual, siguiente, anterior, inicial o final del array, y a la vez actualizan el puntero a esa posición. También disponemos de las funciones *key()*, que devuelve el índice (numérico o asociativo) del elemento actual; y *each()*, que devuelve un array con el índice del elemento actual en las posiciones 0 y 'key', y su valor (lo mismo que devolvería *current*) en las posiciones 1 y 'value', y actualiza el puntero actual al siguiente (lo que haría *next*).

Por ejemplo:

```
$arr = array(1,'cosa',1.57,'gato'=>'raton','perro'=>'gato');

current($arr);    // 1
next($arr);       // cosa
current($arr);    // cosa
prev($arr);       // 1
end($arr);        // gato
current($arr);    // gato
key($arr);        // perro
reset($arr);      // 1
each($arr);       // array(0,1)
each($arr);       // array(1,'foo')
each($arr);       // array(2,1.57)
```

Este puntero al elemento actual del que estamos hablando también se utiliza en los bucles *foreach* y se actualiza en cada iteración.

2.11.4.7 Ordenar un array

En PHP tenemos varias funciones para ordenar un array:

- *sort()*: Ordena el array por contenido en orden ascendente.
- *rsort()*: Ordena por contenido en orden descendente.
- *ksort()*: Ordena por el índice en orden ascendente.
- *rksort()*: Ordena por el índice en orden descendente.

2.11.4.8 Otras funciones útiles

Una operación útil puede ser eliminar los espacios que haya al principio o final de una cadena, por ejemplo si vamos a pasar el texto que ha escrito un usuario en un form a un query en SQL. Esto lo conseguimos con trim:

```
string trim (string cadena)
```

Esta función elimina espacios del principio y final de la cadena. También se puede usar *ltrim* y *rtrim*, que los eliminan sólo del inicio y sólo del final, respectivamente.

Podemos convertir una cadena a mayúsculas con:

```
string strtoupper (string cadena)
```

Y a minúsculas con:

```
string strtolower (string cadena)
```

También puede ser útil convertir a mayúsculas tan sólo el primer carácter de la cadena, por ejemplo si estamos construyendo frases. Esto se consigue con:

```
string ucfirst (string cadena)
```

2.12 Manejo de ficheros

En algunos casos necesitaremos tratar con ficheros y directorios en el sistema local. PHP dispone de toda una serie de funciones para manipularlos, muy similares en sintaxis y uso a las de la librería stdio.h de C.

2.12.1 Abrir y cerrar un fichero

Para abrir un fichero tenemos la función *fopen()*:

```
int fopen (string nombre, string modo [, int include_path])
```

Esta función abre el fichero *nombre* (que puede ser local o remoto, si se indica con "http://" o "ftp://") según *modo*:

- 'r' – Modo de sólo lectura. El puntero se coloca al inicio del fichero.

- 'r+' – Modo lectura/escritura. El puntero se coloca al inicio.
- 'w' – Modo de sólo escritura. Si el fichero existe, se borran sus contenidos. Si no existe, se crea.
- 'w+' – Modo lectura/escritura. Si el fichero existe, se borran sus contenidos. Si no existe, se crea.
- 'a' – Modo sólo escritura. Si el fichero existe, el puntero se coloca al final del fichero. Si no existe se crea.
- 'a+' – Modo lectura/escritura. Se coloca el puntero al final del fichero. Si no existe se crea.

Si el último parámetro se pone a 1, se buscará los ficheros además de en el directorio actual en el indicado en la opción de configuración *include_path* en php.ini.

La función devuelve un identificador para usar con el resto de funciones, o FALSE si ha habido un error.

Para cerrar el fichero, se usa *fclose()* con el identificador devuelto por *fopen()*:

```
int fclose (int identificador)
```

2.12.2 Leer y escribir en el fichero

Se puede leer toda una línea de un fichero con la función *fgets()*:

```
string fgets (int identificador, int tamaño)
```

que devuelve la siguiente línea del fichero apuntado por *identificador*, o *tamaño – 1* caracteres si la línea era más larga.

También se puede leer y a la vez ir procesando la entrada de acuerdo a un formato determinado y almacenándola en variables con la función *fscanf()*, que sigue el mismo formato de *printf*:

```
mixed fscanf (int identificador, string formato
        [, string var1...])
```

Para controlar si hemos llegado al final del fichero, debemos usar *feof()*, que devuelve TRUE si se ha llegado al final o ha habido un error, o FALSE en otro caso:

```
int feof (int identificador)
```

Podemos leer todo el contenido de un fichero con la función *file()*, que no necesita de un *fopen()* y almacena el contenido del fichero leído línea a línea en un array:

```
array file (string fichero [, int include_path])
```

Para escribir en un fichero, usaremos fwrite():

```
int fwrite (int identificador, string cadena [, int tamaño])
```

que escribe en el fichero apuntado por identificador el contenido del string *cadena*, o *tamaño* bytes si *cadena* era más larga.

2.12.3 Copiar / renombrar / borrar ficheros

Podemos copiar un fichero con:

```
int copy (string origen, string destino)
```

renombrarlo (moverlo) con:

```
int rename (string origen, string destino)
```

o borrarlo con:

```
int unlink (string fichero)
```

2.12.4 Directorios

Para movernos a otro directorio, tenemos la función *chdir()*:

```
int chdir (string directorio)
```

Para crear un directorio llamaremos a *mkdir()*:

```
int mkdir (string nombre, int modo)
```

donde modo indica los permisos (en formato UNIX).

Y para borrar directorios:

```
int rmdir (string nombre)
```

Si queremos saber qué ficheros hay dentro de un directorio (como hacer un "dir" en DOS o un "ls" en UNIX), tenemos que usar las funciones *opendir()* para abrirlo, *readdir()* para ir viendo los contenidos y *closedir()* para cerrarlo. Estos son los formatos de estas funciones:

```
int opendir (string nombre)
string readdir (int identificador)
void closedir (int identificador)
```

Por ejemplo, para mostrar la lista de ficheros en el directorio actual, tendríamos que hacer algo así:

```
$direcotrio = opendir('.');
while (($fichero = readdir($directorio)) !== FALSE) {
   echo "$fichero\n";
}
closedir($directorio);
```

2.12.5 Tratamiento de errores

Para evitar que el cliente reciba mensajes de error internos de PHP (errores con la conexión a la base de datos, por ejemplo) se utiliza el operador '@' delante del comando del que queremos ocultar la salida de error. En caso de error deberíamos tomar alguna acción, como por ejemplo detener la ejecución del script y mostrar un mensaje de error nuestro que tenga algún significado para el cliente. La función *die* se encarga de hacer esto:

```
$nombre = '/etc/shadow';
$fichero = @fopen ($nombre, 'r');
if (¡$fichero) {
   die("No se pudo abrir el fichero ($nombre)");
}
```

De esta forma, si *fopen* falla no se muestra en el cliente el mensaje de error que genere, y entonces abortamos la ejecución con la función *die*.

El mensaje de error generado por PHP que ocultamos con el operador '@' se almacena en *$php_errormsg* si la opción *track_errors* ha sido activada en el fichero de configuración.

2.13 include y require

La cláusula *require("fichero");* se sustituye en el código antes de que este se ejecute por el contenido de *fichero*, que puede ser un fichero local o una URL, igual que funciona el *#include* de C o ASP. Esta sustitución se realiza una sola vez, mientras se está preprocesando el contenido del fichero *.php* y antes de ejecutarlo.

La cláusula *include("fichero");* también se sustituye por el contenido de fichero, pero en lugar de realizarse una única vez durante el preproceso del fichero, se realiza durante la ejecución, cada vez que el flujo del programa llega a esa línea.

La utilidad de ambas cláusulas es la misma: imagínese unas líneas de código que se vayan a necesitar en varios ficheros (por ejemplo, una función que valide ciertos datos, o que muestre la cabecera y pie de las páginas). En lugar de copiar ese código en todos los ficheros que lo necesiten, se pone en un fichero que es "incluido" por el resto. Entontes, ¿por qué tener dos funciones distintas para hacer lo mismo? Principalmente por dos motivos:

- Optimización de los accesos a disco: Imagínese un código en el que dependiendo de una serie de condiciones, se deba incluir o no otros ficheros. Con *require* se cargarían

TODOS los ficheros SIEMPRE. Con *include*, únicamente aquellos que se vayan a utilizar.

- Flexibilidad. Con *include* podemos acceder a ficheros cuyo nombre tenemos en una variable que podemos ir cambiando en tiempo de ejecución, mientras que con *require* siempre se accede al mismo fichero.

También existen las funciones *include_once* y *require_once*, que nos aseguran que un determinado fichero sólo será procesado una vez, en caso de que en sucesivos *include*s vuelva a aparecer.

2.14 POO: Clases y Objetos

La idea fundamental de la Programación Orientada a Objetos es la de *encapsular* en una misma entidad (un *objeto*) una serie de datos que definan su estado (*variables de instancia*) y las funciones encargadas de acceder a esos datos (*métodos*.) Las *clases* son las definiciones de la estructura (variables y métodos) de los objetos, y estarían un escalón por arriba de las estructuras simples de los lenguajes de alto nivel (p. Ej. los *struct* de C) y los TADs.

Con esta encapsulación lo que se consigue es que cada objeto de una determinada clase funcione como una *caja negra* de la que poco nos importa su implementación interna. Cuando trabajamos con un objeto de la clase "*coche*" nos da igual cómo funcione internamente el método "*acelerar*". Tan sólo sabemos que nuestro coche puede acelerar, y sabemos cuál será el resultado, sin preocuparnos del proceso interno que se siga para conseguir este fin. De esta forma se independiza la *implementación* de la clase, del *uso* de sus objetos, y el flujo del programa pasa de ser *procedural* a convertirse en un *diálogo entre objetos* que interactúan entre sí.

PHP no es un Lenguaje Orientado a Objetos puro, ya que presenta ciertas carencias, aunque aporta las suficientes características de los LOO para poder hacer un diseño e implementación basado en objetos.

2.14.1 Definición de una clase

La sintaxis para la declaración de clases en PHP es similar a la de C++ o Java:

```
class NombreClase {
   var $variables;

   function metodos ($parametros) {
      codigo
   }
}
```

La definición de las variables dentro de una clase es el único caso en PHP en el que SI que hay que declarar explícitamente una variable antes de usarla, y se hace con la palabra reservada *var*.

Una de las carencias de las clases de PHP es que no se puede restringir el acceso a las variables y métodos. En los LOO puros, hay una serie de palabras reservadas (p.ej., *public*, *private* y *protected* en C++) para indicar si a una variable o método es *público* y se puede acceder desde fuera, o si es *privado* y sólo se puede acceder de forma interna desde otros métodos de la propia clase o sus derivadas (protegido).

Para acceder a las variables y métodos de un objeto, tanto desde fuera como desde un método del propio objeto, hay que utilizar el operador "->", sobre el nombre de la variable que almacena el objeto si accedemos desde fuera (*$miCasa->telefono*), o sobre la referencia al propio objeto *$this* si es desde dentro (*$this->telefono.*)

Existe una clase especial de métodos, llamados *constructores*, que se llaman igual que la clase y se invocan de forma automática al crear un objeto. Estos constructores se encargan de inicializar los valores de las variables internas, reservar memoria, crear estructuras de datos que se vayan a utilizar después... Los constructores pueden tener o no tener parámetros, que habrá que pasar en la creación del objeto.

Como ejemplo, vamos a definir una clase "*coche*":

```
class Coche {
   var $velocidad; // Velocidad actual

   // Constructor por defecto. El coche está parado.
   function Coche() {
      $this->velocidad = 0;
   }

   // Constructor que indica la velocidad inicial.
   function Coche($vel) {
      $this->velocidad = $vel;
   }

   // Método acelerar. El coche va más rápido.
   function acelerar() {
      $this->velocidad++;
   }

   // Método frenar. El coche va más lento hasta frenar.
   function frenar() {
      if ($this->velocidad > 0) {
         $this->velocidad--;
      }
   }
}
```

2.14.2 Herencia

Otra de las características más importantes de la POO junto con la encapsulación es la herencia, por la que se pueden definir clases derivadas de otras ya existentes. Estas clases derivadas (*subclases*) "heredan" todas las características (variables) y funcionalidad (métodos) de sus clases "madres" (*superclases*). Es una forma de ir especializando el tipo de cada clase, partiendo de una clase muy genérica hasta las herederas más específicas.

Para definir una clase en función de otra, se usa la palabra reservada *extends* en la definición de la subclase. De esta forma, la clase heredera parte ya con todas las variables y métodos de su superclase, y además podrá añadir tantas variables y métodos como quiera e incluso redefinir algún método en el proceso de especialización de las clases.

Vamos a definir una subclase de *coche* llamada *cocheFantastico*, que será capaz de hablar, saltar, poner el turbo y que además en lugar de frenar poco a poco frenará de golpe:

```
class CocheFantastico extends coche() {
   // Frenado instantáneo
   function frena() {
      $this->velocidad = 0;
   }

   // ¡El coche habla!
   function habla() {
      echo "Hola, Michael.";
   }

   // ¡Salta!
   function salta() {
      echo "Boing!!";
   }

   // Turbo propulsión
   function turbo() {
      $this->velocidad = 200;
   }
}
```

IMPORTANTE: si se redefinen los constructores, el de la superclase NO SE LLAMA AUTOMÁTICAMENTE al llamar al de la subclase.

2.14.3 Creación y uso de objetos

Los objetos se crean con el operador *new* y el nombre de la clase, y se asignan a una variable para poder usarlos:

```
$MiCoche = new Coche;
```

Si queremos pasarle parámetros a algún constructor, los parámetros se indican entre paréntesis tras el nombre de la clase:

```
$MiCocheSeMueve = new Coche(10);
```

Para acceder a las variables o métodos del objeto, utilizamos el operador "->" sobre el nombre de la variable:

```
$MiCoche->acelerar();
$MiCoche->acelerar();
$MiCoche->acelerar();
echo $MiCoche->velocidad;
$MiCoche->frenar();
```

3 Procesamiento de formularios

3.1 Forms

Trabajar en PHP con *forms* es muy fácil, ya que el propio lenguaje se encarga de crear automáticamente las variables necesarias para almacenar los datos del form en la página que los recibe. Vamos a ver algunos ejemplos:

3.1.1 Valores sencillos

Para los campos de un formulario en los que únicamente se puede elegir o introducir un valor (como campos de texto, desplegables y "radio buttons") PHP crea en el fichero de destino tantas variables como campos haya en el formulario, con los nombres que se les haya dado en éste. Por ejemplo, si tenemos este formulario:

```
<form action="accion.php" method="POST">
Su nombre: <input type=text name=nombre><br>
Su edad: <input type=text name=edad><br>
<input type=submit>
</form>
```

En el script de destino *accion.php*, accederíamos a los valores enviados en el form así:

```
Hola <?=$nombre?>.
Tiene <?=$edad?> años.
```

3.1.2 Valores múltiples

Para los campos de selecciones múltiples PHP también se encarga de almacenar los valores marcados por el usuario en una variable con el nombre del campo, pero en este caso tendremos que llevar cuidado de dar a las variables nombres de *arrays*, ya que si no sólo tendremos acceso al primer valor seleccionado. Por ejemplo este formulario estaría mal:

```
<form action="accion.php" method="POST">
<select multiple name=menu>
<option>Tortilla <option>Paella
<option>Fabada <option>Lentejas
</select><input type=submit></form>
```

Deberíamos haberlo escrito así:

```
<form action="accion.php" method="POST">
<select multiple name=menu[]>
<option>Tortilla <option>Paella
<option>Fabada <option>Lentejas
</select><input type=submit></form>
```

Y podemos ver el resultado con este código:

```
<?php
   echo "Su elección:<br>";
   foreach($menu as $plato) {
      echo "$plato<br>\n";
   }
?>
```

3.2 Cookies

En PHP podemos almacenar datos en una *cookie* utilizando la función *setcookie():*

```
int setcookie (string nombre [, string valor [, int fin
            [, string camino [, string dominio
            [, int seguro]]]]])
```

Como las cookies forman parte de los *headers*, esta función DEBE ser llamada ANTES de realizar cualquier salida de texto, incluso antes de los tags *<html>* y *<head>*. El parámetro fin indica la fecha de expiración de la cookie y se expresa en formato UNIX (como las funciones de PHP *time()* y *mktime()*). El parámetro *seguro* fuerza a que la cookie sólo sea enviada a través de una conexión segura (HTTPS). Los campos optativos de tipo string se pueden omitir poniendo "", y los numéricos con "0".

Por ejemplo, para definir una cookie llamada "PruebaCookie" con el valor "expiraré dentro de una hora" y que, en efecto, expire en una hora, tendríamos que hacer:

```
setcookie("PruebaCookie", "expiraré dentro de una hora",
        time() + 3600);
```

La próxima vez que se cargue la página, PHP definirá una variable global llamada "PruebaCookie" con el valor definido para ésta.

Para borrar la cookie del ejemplo anterior (hay que indicar los mismos parámetros que se usaron para definirla):

```
setcookie("PruebaCookie", "", time());
```

3.3 Sesiones

Para iniciar el seguimiento de una *sesión*, llamamos a la función *session_start()*:

```
bool session_start(void);
```

Si ya había una sesión (almacenada en una cookie o pasada por GET), la continúa. Si no, crea una nueva.

Una vez que hemos iniciado el tratamiento de sesiones, podemos registrar variables en la sesión con *session_register()*:

```
bool session_register (mixed name [, mixed ...])
```

Podemos registrar en una llamada a *session_register()* tantas variables como queramos.

Con estas funciones tenemos la misma restricción que con las cookies: deben llamarse antes de realizar cualquier tipo de salida de texto.

Por ejemplo, para implementar un contador del número de veces que un usuario visita una página, podríamos hacer:

```
session_start();
print($contador);
$contador++;
session_register("contador");
```

Para finalizar una sesión, llamamos a *session_destroy()*:

```
bool session_destroy(void);
```

Esta función finaliza la sesión actual, es decir, debe ser llamada siempre después de *session_start()*.

El identificador de la sesión se pasa por defecto mediante cookies. Si estas están desactivadas en el cliente, tendremos que pasarlo como una variable GET. PHP define la constante SID con el Session ID actual, de forma que para pasarlo por GET podemos hacer:

```
<A HREF="siguiente_pagina.php?<?=SID?>">Continuar</A>
```

4. PostgreSQL.

SQL es el manejador de base de datos que soportan casi todos los sistemas de bases de datos.

4.1. Creando tablas.

El formato del comando para crear una tabla es el siguiente:

CREATE TABLE nom_tabla(
Nom_columna1 tipo(tamaño),
Nom_columna2 tipo(tamaño),
.
.
.
Nom_columnaN tipo(tamaño),
);

Para desplegar las características de una tabla específica, sólo se tiene que teclear *\d nom_tabla.*

4.2. Agregando datos.

El formato del comando para agregar datos a una tabla es el siguiente:

INSERT INTO nom_tabla VALUES(
'dato_1',
'dato_2',
.
.
.
'dato_N'
);

Si el dato es un número, no es necesario poner las comillas simples. Por otra parte, cada uno de los datos se tiene que meter en el orden en que se encuentran las columnas de la tabla. Otra manera de insertar los datos, es agregándolos únicamente a ciertas columnas, como se muestra en el siguiente formato:

INSERT INTO nom_tabla (nom_col2, nomcol5,..., nom_colX) VALUES(dato_2, dato_5, ...,dato_X);

Si lo que se desea es insertar un valor nulo de manera directa se puede hacer con la palabra reservada *NULL*; y para saber si un valor es nulo, se usa la función *IS NULL*.

Nota:

Con *NULL* se inserta un valor nulo en el campo, que es diferente de cero(entero 0) o carácter cero('0'), o de un texto vacío(' '). Por lo cual se debe de tener cuidado. Otro aspecto importante es que no se pueden hacer comparaciones entre campos que tengan

NULL; es decir, si la columna edad tiene un *NULL* en un renglón y la columna ciudad tiene *NULL*, y queremos desplegar aquellos renglones en que ciudad y edad sean iguales, en el resultado no aparecerán los renglones en los que ciudad y edad sean *NULL*.

4.3. Visualizando datos.

El formato del comando para visualizar datos de una tabla es el siguiente:
*SELECT * FROM nom_tabla;*

De esta forma, se desplegarán todos los renglones de la tabla especificada, de cada una de las columnas. Otra forma de desplegar la información de la tabla, es seleccionando la o las columnas que se desean desplegar; esto se puede hacer de la siguiente manera:
SELECT nom_columna1,nom_columna2,...,nom_columnaN FROM nom_tabla;

De esta forma, se puede desplegar desde una sola columna hasta todas las de la tabla, mediante la separación con la (,).

4.4. Seleccionando renglones específicos.

El formato del comando para visualizar datos específicos de una tabla es el siguiente:
SELECT nom_col1,nom_col2,...,nom_colN FROM nom_tabla WHERE nom_colX(=,<,>)nom_colXN;

Ejemplos:
*SELECT * FROM Amigo WHERE edad=23;*
SELECT ciudad FROM Amigo WHERE edad<=20;
SELECT apellido FROM Amigo WHERE nombre='Pérez';
SELECT nacionalidad,calle FROM Amigo WHERE edad>=27;

4.5. Borrando datos.

El formato del comando para borrar datos de una tabla es el siguiente:
DELETE FROM nom_tabla;

De esta forma se borrarán todos los datos de la tabla. Otra manera de hacerlo es, eligiendo los renglones que se desean borrar, lo cual se puede hacer de la siguiente forma:
DELETE FROM nom_tabla WHERE nom_colX(=,<,>=)'X';

Con lo cual sólo se borrarán los renglones que coincidan con la condición.

4.6. Modificando datos.

El formato del comando para modificar o actualizar datos de una tabla es el siguiente:

UPDATE nom_tabla SET nom_colX(<=,=)'X' WHERE nom_col(=,<,)'Z';

El comando SET lo que indica es la columna que se cambiará y el valor que tomará al encontrar el dato.

4.7. Acomodando datos.

El formato del comando para acomodar datos de una tabla es el siguiente:

*SELECT * FROM nom_tabla ORDER BY nom_colX;*

De esta forma los renglones se ordenarán de manera ascendente, de acuerdo a la columna seleccionada. Otra forma de ordenar los datos es hacerlo de manera descendente, como se muestra en el siguiente formato:

*SELECT * FROM nom_tabla ORDER BY nom_colX DESC;*

4.8. Destruyendo tablas.

El formato del comando para destruir una tabla es el siguiente:

DROP TABLE nom_tabla;

Este comando además de borrar los datos de la tabla, destruye totalmente la estructura de la misma.

4.9. Ejemplos.

*FAMILIA=>SELECT * FROM Amigo;*

Nombre	Apellido	Estado	edad
Alejandro	Reyes	DF	22
Vanessa	Soto	Estado de México	16
Claudia	Mena	Sonora	23
Blanca	Pérez	Zacatecas	12
Jorge	Hernández	Chiapas	58

(5 rows)

FAMILIA=>INSERT INTO Amigo VALUES('Juan','Salinas','Oaxaca',89);

INSERT 18880 1

*FAMILIA=> SELECT * FROM Amigo;*

Nombre	Apellido	Estado	edad
Alejandro	Reyes	DF	22
Vanessa	Soto	Estado de México	16
Claudia	Mena	Sonora	23
Blanca	Pérez	Zacatecas	12
Jorge	Hernández	Chiapas	58
Juan	Salinas	Oaxaca	89

(6 rows)

FAMILIA=> DELETE FROM Amigo WHERE apellido='Soto';
DELETE 1

*FAMILIA=> SELECT * FROM Amigo;*

Nombre	Apellido	Estado	edad
Alejandro	Reyes	DF	22
Claudia	Mena	Sonora	23
Blanca	Pérez	Zacatecas	12
Jorge	Hernández	Chiapas	58
Juan	Salinas	Oaxaca	89

(5 rows)

FAMILIA=>UPDATE Amigo SET edad=21 WHERE nombre='Claudia';
UPDATE 1

*FAMILIA=> SELECT * FROM Amigo;*

Nombre	Apellido	Estado	edad
Alejandro	Reyes	DF	22
Claudia	Mena	Sonora	21
Blanca	Pérez	Zacatecas	12
Jorge	Hernández	Chiapas	58
Juan	Salinas	Oaxaca	89

(5 rows)

*FAMILIA=> SELECT * FROM Amigo ORDER BY estado;*

Nombre	Apellido	Estado	edad
Jorge	Hernández	Chiapas	58
Alejandro	Reyes	DF	22
Juan	Salinas	Oaxaca	89
Claudia	Mena	Sonora	21
Blanca	Pérez	Zacatecas	12

(5 rows)

*FAMILIA=> SELECT * FROM Amigo ORDER BY edad DESC;*

Nombre	Apellido	Estado	edad
Juan	Salinas	Oaxaca	89
Jorge	Hernández	Chiapas	58
Alejandro	Reyes	DF.	22
Claudia	Mena	Sonora	21
Blanca	Pérez	Zacatecas	12

(5 rows)

*FAMILIA=> SELECT * FROM Amigo WHERE edad>=25 ORDER BY nombre;*

Nombre	Apellido	Estado	edad
Jorge	Hernández	Chiapas	58
Juan	Salinas	Oaxaca	89

(2 rows)

5. PHP y PostgreSQL.

5.1. Funciones relacionadas.

pg_Conect().

Esta función nos permite conectarnos con la base de datos. Recibe una serie de parámetros, los cuales pueden o no ir, ya que PHP se encarga de la verificación de los datos. La cantidad de parámetros es variable, sin embargo, la más usada es la que presenta cinco parámetros. El primero representa el host en donde se encuentra la base de datos; el segundo, es el número de puerto habilitado para el proceso; el tercero, es de opciones, entre las cuales puede estar el usuario y la contraseña, el cuarto es la terminal y el quinto es el nombre de la base de datos a la que se conectará. Su formato es:
int **pg_connect** (string host, string port, string options, string tty, string dbname)

pg_Dbname().

Nos permite obtener el nombre de la base de datos a la que se tiene conexión. Recibe como parámetro la clave con la que se realizó la conexión. Su formato es:
string **pg_dbname** (int connection)

pg_Exec().

Ejecuta un query, es decir, una petición hacia la base de datos y regresa un apuntador al resultado. Su formato es:
int **pg_exec** (int connection, string query)

pg_NumRows().

Obtiene el número de renglones del resultado del query. Su formato es:
int **pg_numrows** (int result_id)

pg_FieldName().

Obtiene el nombrede un campo específico.Su formato es:
string **pg_fieldname** (int result_id, int field_number)

pg_Result().

Regresa los resultados del query a partir del identificador. Su formato es:
mixed **pg_result** (int result_id, int row_number, mixed fieldname)

pg_FieldPrtlen().

Obtiene el tamaño de la cadena impresa. Su formato es:
int **pg_fieldprtlen** (int result_id, int row_number, string field_name)

pg_FieldSize().

Obtiene el tamaño del campo. Su formato es:
int **pg_fieldsize** (int result_id, int field_number)

pg_FieldType().

Obtiene el tipo del campo especificado. Su formato es:
int **pg_fieldtype** (int result_id, int field_number)

pg_ErrorMessage().

Regresa el error ocurrido durante la conexión a la base de datos. Su formato es:
string **pg_errormessage** (int connection)

pg_Fetch_Array().

Devuelve el resultado de un query en forma de arreglo. Su formato es:
array **pg_fetch_array** (int result, int row [, int result_type])

pg_FieldNum().

Regresa el número de campo. Su formato es:
int **pg_fieldnum** (int result_id, string field_name)

pg_FieldIsNull().

Indica si el contenido del campo es nulo. Su formato es:
int **pg_fieldisnull** (int result_id, int row, mixed field)

pg_Close().

Cierra la conexión con la base de datos. El formato de la función es:
bool **pg_close** (int connection)

5.2. Ejemplo.

```
<HTML>
<BODY>
<?php
 error_reporting(1);
$database=pg_Connect("localhost","5432","PRUEBA");
// $database=pg_Connect("","","PRUEBA");
// $database=pg_Connect("","","","","PRUEBA");

if(!$database)
{
```

```
   echo "Falló la conexión.";
   exit;
}

$nombre=pg_Dbname($database);
print $nombre."\n<BR>";
$result=pg_Exec($database,"SELECT nombre,codigo FROM Estado WHERE
codigo=210");

for($i=0;$i<pg_NumRows($result);$i++)
{
   echo pg_Fieldname($result,0)." ";
   echo pg_Result($result,$i,0);
   echo "<BR>\n";
   echo pg_Fieldname($result,1)." ";
   echo pg_Result($result,$i,1);
   echo "<BR>\n";
   $long=pg_Fieldprtlen($result,$i,"nombre");
   echo $long."\n<BR>";
   $tam=pg_Fieldsize($result,0);
   echo $tam."\n<BR>";
   $tipo=pg_Fieldtype($result,0);
   echo $tipo."\n<BR>";
}
$error=pg_Errormessage($database);
print "<BR>\n$error";

echo "<BR>\n<BR>\n<BR>\n";
$arreglo = pg_fetch_array ($result, 0);
echo "$arreglo[0]\n<BR>";
$arreglo = pg_fetch_array ($result, 1);
echo $arreglo["nombre"]."\n<BR>";

echo "<BR>\n<BR>\n<BR>\n";
$num=pg_fieldnum($result,"codigo");
echo $num;

echo "<BR>\n<BR>\n<BR>\n";
$nulo=pg_Fieldisnull($result,10,1);
echo $nulo."\n<BR>";

echo "<BR>\n<BR>\n<BR>\n";
$long=pg_Fieldprtlen($result,7,"nombre");
echo $long.$result."\n<BR>";

pg_close($database);
?>
</BODY>
```

</HTML>

Créditos

Mora Cuevas Jonathan.
Reyes Sáenz Hugo Alejandro