

PHP

(Gabriele Farina, Gianluca Gillini, Gabriele Tassoni)

Introduzione

PHP è un linguaggio di scripting interpretato, con licenza open source, originariamente concepito per la realizzazione di pagine web dinamiche. Attualmente è utilizzato principalmente per sviluppare applicazioni web lato server ma può essere usato anche per scrivere script a linea di comando o applicazioni standalone con interfaccia grafica.

Il suo nome è un acronimo ricorsivo che sta per PHP: Hypertext Preprocessor (PHP: preprocessore di ipertesti).

A metà degli anni Novanta il Web era ancora formato in gran parte da pagine statiche, cioè da documenti HTML il cui contenuto non poteva cambiare fino a quando qualcuno non interveniva manualmente a modificarlo. Con l'evoluzione di Internet, però, si cominciò a sentire l'esigenza di rendere dinamici i contenuti, cioè di far sì che la stessa pagina fosse in grado di proporre contenuti diversi, personalizzati in base alle preferenze degli utenti, oppure estratti da una base di dati (database) in continua evoluzione.

PHP nasce nel 1994, ad opera di Rasmus Lerdorf, come una serie di macro la cui funzione era quella di facilitare ai programmatori l'amministrazione delle homepage personali: da qui trae origine il suo nome, che allora significava appunto Personal Home Page. In seguito, queste macro furono riscritte ed ampliate fino a comprendere un pacchetto chiamato Form Interpreter (PHP/FI).

Essendo un progetto di tipo open source (cioè "codice aperto", quindi disponibile e modificabile da tutti), ben presto si formò una ricca comunità di sviluppatori che portò alla creazione di PHP 3: la versione del linguaggio che diede il via alla crescita esponenziale della sua popolarità. Tale popolarità era dovuta anche alla forte integrazione di PHP con il Web server Apache (il più diffuso in rete), e con il database MySQL. Tale combinazione di prodotti, integralmente ispirata alla filosofia del free software, diventò ben presto vincente in un mondo in continua evoluzione come quello di Internet.

Alla fine del 1998 erano circa 250.000 (10% market share) i server Web che supportavano PHP: un anno dopo superavano il milione. I 2 milioni furono toccati in aprile del 2000, e alla fine dello stesso anno erano addirittura 4.800.000. Il 2000 è stato sicuramente l'anno di maggiore crescita del PHP, coincisa anche con il rilascio della versione 4, con un nuovo motore (Zend) molto più veloce del precedente ed una lunga serie di nuove funzioni, fra cui quelle importantissime per la gestione delle sessioni. La crescita di PHP, nonostante sia rimasta bloccata fra luglio e ottobre del 2001, è poi proseguita toccando quota 7.300.000 server alla fine del 2001, per superare i 10 milioni alla fine del 2002, quando è stata rilasciata la versione 4.3.0. Nel 2005 il market share raggiunto era del 65% nella configurazione che lo vede installato assieme a MySQL, Apache e Linux, chiamata LAMP. La continua evoluzione dei linguaggi di programmazione concorrenti e l'incremento notevole dell'utilizzo del linguaggio anche in applicazioni enterprise ha portato la Zend a sviluppare una nuova versione del motore per supportare una struttura ad oggetti molto più rigida e potente.

Nasce così PHP 5, che si propone come innovazione nell'ambito dello sviluppo web open source soprattutto grazie agli strumenti di supporto professionali forniti con la distribuzione standard ed al grande sforzo di Zend che, grazie alla partnership con IBM, sta cercando di spingere sul mercato soluzioni di supporto enterprise a questo ottimo linguaggio. Lo sviluppo di PHP procede comunque con due progetti paralleli che supportano ed evolvono sia la versione 4 che la versione 5. Questa scelta è stata fatta poichè tuttora sono pochi i fornitori di hosting che hanno deciso di fare il porting dei propri server alla nuova versione del linguaggio.

Oggi PHP è conosciuto come Hypertext Preprocessor, ed è un linguaggio completo di scripting, sofisticato e flessibile, che può girare praticamente su qualsiasi server Web, su qualsiasi sistema operativo (Windows o Unix/Linux, ma anche Mac, AS/400, Novell, OS/2 e altri), e consente di interagire praticamente con qualsiasi tipo di database (SQLite, MySQL, PostgreSQL, SQL Server, Oracle, SyBase, Access e altri). Si può utilizzare per i più svariati tipi di progetti, dalla semplice home page dinamica fino al grande portale o al sito di e-commerce.

La programmazione lato server e lato client

Parlando di PHP e di altri linguaggi di scripting può capitare di sentir citare le espressioni "lato client" e "lato server": per chi non è esperto della materia, tali definizioni possono suonare un po' misteriose. Proviamo a chiarire questi concetti: vediamo come funziona, in maniera estremamente semplificata, la richiesta di una pagina Web. L'utente apre il suo browser e digita un indirizzo Internet, ad esempio `www.nostrosito.it/pagina1.html`: a questo punto il browser si collega al server `www.nostrosito.it` e gli chiede la pagina `pagina1.html`. Tale pagina contiene esclusivamente codice HTML: il server la prende e la spedisce al browser, così com'è (insieme ad eventuali file allegati, ad esempio immagini). Il nostro utente quindi avrà la possibilità di visualizzare questa pagina.

Supponiamo ora che l'utente richieda invece la pagina `pagina2.php`: questa, contrariamente a quella di prima, non contiene solo codice HTML, ma anche PHP. In questo caso il server, prima di spedire la pagina, esegue il codice PHP, che in genere produce altro codice HTML: ad esempio, PHP potrebbe controllare che ore sono e generare un messaggio di questo tipo: "Buon pomeriggio, sono le 17.10!" oppure: "Ehi, che ci fai alzato alle 4 del mattino?". Dopo l'esecuzione, la pagina non conterrà più codice PHP, ma solo HTML. A questo punto è pronta per essere spedita al browser. (Ovviamente, il file che non contiene più codice PHP non è quello "originale", ma la "copia" che viene spedita al browser. L'originale rimane disponibile per le prossime richieste.) Quindi l'utente vede solo il codice HTML, e non ha accesso al codice PHP che ha generato la pagina.

Per comprendere ancora meglio questo concetto, confrontiamo PHP con un altro linguaggio di scripting molto diffuso sul Web, cioè JavaScript, che di solito viene usato come linguaggio "lato client": JavaScript infatti viene eseguito non dal server, ma dal browser dell'utente (il client, appunto). JavaScript ci consente di eseguire operazioni che riguardano il sistema dell'utente, come ad esempio aprire una nuova finestra del browser, o controllare la compilazione di un modulo segnalando eventuali errori prima che i dati vengano spediti al server. Ci permette anche di avere un'interazione con l'utente: ad esempio, possiamo far sì che quando il mouse passa su una determinata immagine, tale immagine si modifichi.

Per svolgere tutti questi compiti, JavaScript deve essere eseguito sul sistema dell'utente: per questo il codice JavaScript viene spedito al browser insieme al codice HTML. Quindi l'utente ha la possibilità di visualizzarlo, contrariamente a ciò che accade con PHP. Abbiamo citato alcune utili funzioni svolte da JavaScript sul browser dell'utente: PHP, essendo eseguito sul server, non è in grado di svolgere direttamente queste funzioni. Ma attenzione: questo non significa che non sia in grado ugualmente di controllarle! Infatti PHP svolge principalmente la funzione di 'creare' il codice della pagina che viene spedita all'utente: di conseguenza, così come può creare codice HTML, allo stesso modo può creare codice JavaScript. Questo significa che PHP ci può permettere, ad esempio, di decidere se ad un utente dobbiamo spedire il codice JavaScript che apre una nuova finestra, oppure no. In pratica, quindi, lavorando sul lato server abbiamo il controllo anche del lato client. Rimane un ultimo dettaglio da svelare: come fa il server a sapere quando una pagina contiene codice PHP che deve essere eseguito prima dell'invio al browser? Semplice: si basa sull'estensione delle pagine richieste.

Nell'esempio che abbiamo visto prima, `pagina1` aveva l'estensione `.html`, mentre `pagina2` aveva l'estensione `.php`: sulla base di questo, il server sa che nel secondo caso deve eseguire PHP, mentre nel primo può spedire il file così com'è. In realtà il server deve essere istruito per poter fare ciò: generalmente gli si dice di eseguire PHP per le pagine che hanno estensione `.php`. È possibile comunque assegnargli qualsiasi altra estensione (fino a qualche anno fa veniva utilizzata `phtml`, anche se ormai la pratica è caduta in disuso). Si possono utilizzare anche le estensioni standard `.htm` e `.html`, ma ciò significherebbe chiamare PHP per tutte le pagine richieste, anche se non contengono codice PHP: questo rallenterebbe inutilmente il lavoro del server e dunque è meglio evitarlo.

Caratteristiche di PHP

A fronte di quello detto precedentemente va precisato che PHP non è l'unico linguaggio lato server disponibile per chi si appresta a sviluppare pagine web. Sono disponibili varie alternative, sia proprietarie che open source, ed ognuna di queste ha i suoi pregi ed i suoi difetti. Dato che il paragone tra due linguaggi di programmazione di buon livello è spesso soggettivo, in questa sede

preferisco descrivere in modo semplice le caratteristiche di PHP ed i vantaggi che queste possono portare allo sviluppatore, lasciando a voi l'eventuale compito di confrontarlo con altri strumenti.

La versione di PHP su cui si basa questa guida (la 5.1.2) implementa soluzioni avanzate che permettono un controllo completo sulle operazioni che possono essere svolte dal nostro server web. L'accesso ai cookie ed alle sessioni è molto semplice ed intuitivo, avvenendo attraverso semplici variabili che possono essere accedute da qualunque posizione all'interno del codice.

PHP riprende per molti versi la sintassi del C, come peraltro fanno molti linguaggi moderni, e del Perl. È un linguaggio a tipizzazione debole e dalla versione 5 migliora il supporto al paradigma di programmazione ad oggetti. Certi costrutti derivati dal C, come gli operatori fra bit e la gestione di stringhe come array, permettono in alcuni casi di agire a basso livello; tuttavia è fondamentalmente un linguaggio di alto livello, caratteristica questa rafforzata dalla esistenza delle sue moltissime API, oltre 3000 funzioni del nucleo base. PHP è in grado di interfacciarsi a innumerevoli database tra cui MySQL, PostgreSQL, Oracle, Firebird, IBM DB2, Microsoft SQL Server, solo per citarne alcuni, e supporta numerose tecnologie, come XML, SOAP, IMAP, FTP, CORBA. Si integra anche con altri linguaggi/piattaforme quali Java e .NET e si può dire che esista un wrapper per ogni libreria esistente, come CURL, GD, Gettext, GMP, Ming, OpenSSL ed altro.

Fornisce un'API specifica per interagire con Apache, nonostante funzioni naturalmente con numerosi server web. È anche ottimamente integrato con il database MySQL, per il quale possiede più di una API. Per questo motivo esiste un'enorme quantità di script e librerie in PHP, disponibili liberamente su Internet. La versione 5, comunque, integra al suo interno un piccolo database embedded, SQLite.

PHP ha una lunga storia legata esclusivamente al web, e per questo motivo esistono moltissime librerie testate e complete per svolgere i compiti più diversi: esistono strumenti per la gestione delle template, librerie che permettono la gestione completa di un mail server, sia in invio che in ricezione e molto altro ancora. A supporto di tutto questo bisogna dire che il modulo per eseguire script PHP è ormai installato di default sui server di hosting, e che la comunità di sviluppatori risolve molto velocemente i bug che si presentano agli utenti.

A supporto di PHP, sia Zend che la comunità php.net, hanno associato una serie di strumenti molto utili:

- Il repository PEAR che contiene decine di classi ben organizzate e documentate per svolgere la maggior parte delle operazioni ad alto e basso livello richieste durante lo sviluppo di applicazioni web. Tra queste ricordiamo il layer di astrazione per l'accesso ai database, le classi per il debugging ed il logging, quelle per la generazione di grafici avanzati e quelle per la gestione dei template.
- Il repository PECL che contiene molte estensioni native che estendono le potenzialità del linguaggio con funzionalità di basso livello ad alte prestazioni. Abbiamo sistemi di cache ed ottimizzazione del codice intermedio generato durante l'esecuzione di script PHP, sistemi per il debugging avanzato ed il profiling del codice e molto altro.
- Il template engine Smarty, uno dei più robusti ed utilizzati template engine per PHP in circolazione.

A tutto questo va aggiunto che la collaborazione di Zend con IBM sta portando allo sviluppo di strumenti di supporto professionali per gli sviluppatori, quali Zend Studio 5.0, Zend Safe Guard ed altri strumenti che coprono perfettamente tutto il processo di sviluppo e mantenimento del software. Non è tutto rose e fiori purtroppo: per esempio il fatto che PHP venga distribuito in due versioni differenti (tuttora la 4 e la 5) limita gli sviluppatori nell'utilizzo delle caratteristiche della nuova versione, dato che la maggior parte dei servizi di hosting continuano ad aggiornare quella precedente. Oltretutto il fatto che PHP funzioni ad estensioni non è sempre un vantaggio, dato che spesso e volentieri i vari servizi di hosting hanno configurazioni differenti. Un'altra lacuna, che obbliga gli sviluppatori a sviluppare codice specifico per aggirarla, è la mancanza del supporto per caratteri Unicode che rende più complicato lo sviluppo di applicazioni multilingua per paesi che accettano caratteri speciali all'interno delle loro parole. Fortunatamente la versione 6 di PHP (che è tutt'ora in sviluppo) includerà nativamente questo supporto.

Insomma: PHP è un ottimo linguaggio, leader tra quelli open source per lo sviluppo web, molto semplice da imparare e subito produttivo. Oltretutto ha una serie di strumenti di appoggio molto

completi e con la versione 5 un robusto supporto per la programmazione ad oggetti. Anche se con qualche difetto, penso sia la scelta più adeguata per un gran numero di situazioni.

PHP e HTML

PHP è un linguaggio la cui funzione fondamentale è quella di produrre codice HTML, che è quello dal quale sono formate le pagine web. Ma, poichè PHP è un linguaggio di programmazione, abbiamo la possibilità di analizzare diverse situazioni (l'input degli utenti, i dati contenuti in un database) e di decidere, di conseguenza, di produrre codice HTML condizionato ai risultati dell'elaborazione. Questo è, in parole povere, il Web dinamico. Come abbiamo visto precedentemente, quando il server riceve una richiesta per una pagina PHP, la fa analizzare dall'interprete del linguaggio, il quale restituisce un file contenente solo il codice che deve essere inviato al browser (in linea di massima HTML, ma può esserci anche codice JavaScript, fogli di stile CSS o qualunque altro contenuto fruibile da un browser, come immagini e documenti Pdf).

Detto questo, come avviene la produzione di codice HTML? La prima cosa da sapere è come fa l'interprete PHP a discernere quale porzione di un file contiene codice da elaborare e quale codice da restituire solamente all'utente. Questa fase di riconoscimento è molto importante, dato che permette a PHP di essere incluso all'interno di normale codice HTML in modo da renderne dinamica la creazione. Il codice PHP deve essere compreso fra appositi tag di apertura e di chiusura, che sono i seguenti:

```
<?php //tag di apertura  
?> //tag di chiusura
```

Tutto ciò che è contenuto fra questi tag deve corrispondere alle regole sintattiche del PHP, ed è codice che sarà eseguito dall'interprete e non sarà inviato direttamente al browser al browser. Per generare l'output da inviare al browser attraverso codice PHP viene normalmente utilizzato il costrutto echo. Vediamo un semplice esempio, composto da codice HTML e codice PHP (il codice PHP è evidenziato in rosso):

```
<html>  
  <head>  
    <title>  
    <?php  
      echo "Pagina di prova PHP";  
    ?>  
    </title>  
  </head>  
  <body>  
  <?php  
    echo "Buona giornata!";  
  ?>  
  </body>  
</html>
```

Questo banalissimo codice produrrà un file HTML il cui contenuto sarà semplicemente:

```
<html>  
  <head>  
    <title>  
      Pagina di prova PHP  
    </title>  
  </head>  
  <body>  
    Buona giornata!  
  </body>
```

```
</html>
```

E quindi l'utente vedrà sul suo browser la riga "Buona giornata!". È opportuno ricordare che il dato da inviare al browser che segue il comando echo può essere racchiuso tra parentesi e che al comando possono essere date in input più stringhe (questo è il nome che viene dato ad una ripetizione di qualunque carattere compreso tra due apici singoli (' ') o doppi (" ")), separate da virgole, così:

```
echo "Buongiorno a tutti!", "<br />\n", "È una bellissima giornata";
```

Se si decide di utilizzare il separatore virgola, non possono essere utilizzate le parentesi. Nel prosieguo del corso useremo spesso il verbo 'stampare' riferito alle azioni prodotte dal comando echo o da istruzioni con funzionalità analoghe (quali print, sprintf e altro): ricordiamoci però che si tratta di una convenzione, perchè in questo caso la 'stampa' non avviene su carta, ma sull'input che verrà inviato al browser!

Facciamo caso ad un dettaglio: nelle istruzioni in cui stampavamo "Buongiorno a tutti", abbiamo inserito, dopo il
, il simbolo \n. Questo simbolo ha una funzione abbastanza importante nella programmazione e nello scripting che serve più che altro per dare leggibilità al codice HTML che stiamo producendo. Infatti PHP, quando trova questa combinazione di caratteri fra virgolette, li trasforma in un carattere di ritorno a capo: questo ci permette di controllare l'impaginazione del nostro codice HTML. Bisogna però stare molto attenti a non confondere il codice HTML con il layout della pagina che l'utente visualizzerà sul browser: infatti, sul browser è solo il tag
 che forza il testo ad andare a capo.

Quando questo tag non c'è, il browser allinea tutto il testo proseguendo sulla stessa linea (almeno fino a quando gli altri elementi della pagina e le dimensioni della finestra non gli "consigliano" di fare diversamente), anche se il codice HTML ha un ritorno a capo.

Vediamo di chiarire questo concetto con un paio di esempi:

```
<?php  
echo "prima riga\n";  
echo "seconda riga<br />";  
echo "terza riga";  
?>
```

Questo codice php produrrà il seguente codice HTML:

```
prima riga  
seconda riga<br />terza riga
```

mentre l'utente, sul browser, leggerà:

```
prima riga seconda riga  
terza riga
```

Questo perchè il codice PHP, mettendo il codice 'newline' dopo il testo 'prima riga', fa sì che il codice HTML venga formato con un ritorno a capo dopo tale testo. Il file ricevuto dal browser quindi andrà a capo proprio lì. Il browser, però, non trovando un tag che gli indichi di andare a capo, affiancherà la frase 'prima riga' alla frase 'seconda riga', limitandosi a mettere uno spazio fra le due. Successivamente accade l'esatto contrario: PHP produce un codice HTML nel quale il testo 'seconda riga' è seguito dal tag
, ma non dal codice 'newline'. Per questo, nel file HTML, 'seconda riga
' e 'terza riga' vengono attaccati. Il browser, però, quando trova il tag
 porta il testo a capo.

Avrete forse notato che in fondo ad ogni istruzione PHP abbiamo messo un punto e virgola; infatti la sintassi del PHP prevede che il punto e virgola debba obbligatoriamente chiudere ogni istruzione. Ricordiamoci quindi di metterlo sempre, con qualche eccezione che vedremo più avanti. Da quanto abbiamo detto finora emerge una realtà molto importante: chi vuole avvicinarsi al PHP deve già

avere una conoscenza approfondita di HTML e di tutto quanto può far parte di una pagina web. Questo perchè lo scopo principale di PHP è proprio la produzione di questi codici (anche se va ricordato che può essere utilizzato per scopi differenti, come linguaggio di shell o per la creazione di applicazioni desktop grazie all'estensione PHP-GTK).

I commenti

Un altro argomento molto importante legato alla sintassi di PHP sono i commenti. Chi ha esperienza, anche minima, di programmazione, sa bene che molte volte i commenti si rivelano di importanza decisiva quando si tratta di mettere le mani su un programma realizzato da qualcun altro, e anche quando il programma è stato scritto da noi stessi, soprattutto se è passato qualche tempo dalla realizzazione. I commenti svolgono un ruolo fondamentale in questa fase di "rivisitazione" del codice, in quanto possono facilitare di molto la comprensione di passaggi apparentemente oscuri.

È bene quindi non risparmiare mai un commento quando possibile (senza comunque esagerare altrimenti, al posto di chiarire il codice lo renderete ancora più illeggibile), anche perchè il loro utilizzo non appesantisce l'esecuzione dello script (l'interprete PHP salta tutte le parti che riconosce come commenti), nè il trasferimento della pagina al browser (infatti i commenti, essendo contenuti all'interno del codice PHP, fanno parte di ciò che non viene inviato al browser).

Abbiamo tre diverse possibilità per posizionare i commenti all'interno del nostro codice: la prima è l'uso dei commenti in stile C++, caratterizzati da due barre:

```
<?php
// Commento in stile C++
?>
```

La seconda sono i commenti in stile Perl e Python, contraddistinti dall'uso del cancelletto (anche se ormai obsoleti e poco utilizzati):

```
<?php
# Commento in stile Perl
# e python
?>
```

Entrambi questi tipi di commenti sono limitati ad una sola riga: l'interprete PHP, quando trova le barre o il cancelletto, salta tutto ciò che si trova da quel punto fino al termine della riga. Questo ci permette di porre il commento anche sulla stessa riga del codice commentato, così:

```
<?php
echo 'Buongiorno a tutti <br />';
//stampo un messaggio di saluto
print 'Esclusi quelli antipatici';
# faccio una precisazione
// Questa riga contiene solo commento
?>
```

L'ultimo tipo di commento che abbiamo a disposizione permette di specificare commenti multilinea senza dover ripetere i caratteri ogni nuova riga grazie ad una coppia di caratteri utilizzati per l'apertura e la chiusura. Tutto il codice che segue /* viene considerato commento da PHP finchè non incontra la serie di caratteri */. Un semplice esempio:

```
<?php

/*
Questo è un commento
multiriga specificando
utilizzando la stessa sintassi
*/
```

```
usata in Java e C
*/
echo /* commento */ "Ciao a tutti" /* i commenti vengono saltati*/;

?>
```

Riguardo i commenti multilinea, è importante ricordare che non possono essere innestati e che comunque rappresentano un separatore per l'interprete PHP. Per questo motivo le seguenti sintassi sono errate:

```
<?php
/*
Commento /*
multilinea
*/
Qui verrà generato un errore ...
*/

ec/* questa sintassi è errata */ho "prova";

?>
```

La scelta di quale tipo di commento utilizzare è solitamente soggettiva, anche se spesso vengono utilizzati i commenti multiriga per documentare il codice e quelli a riga singola per aggiungergli dei semplici appunti sul funzionamento logico.

Le variabili

Come per tutti i linguaggi di programmazione anche con il PHP è possibile utilizzare le variabili, che sono rappresentate dal simbolo del dollaro (\$) seguito dal nome della variabile.

```
$variabile = 1;
$Variabile = "Questa è una variabile";
```

Queste sono variabili, e sono differenti non tanto per il valore loro assegnato quanto per il loro nome: infatti, in PHP le variabili sono case-sensitive.

Ogni variabile in PHP può assumere un nome che inizia con una lettera dell'alfabeto o un underscore (_) e segue con una combinazione qualsiasi di lettere, numeri o underscore. È bene ricordarsi che PHP, durante il processo di inizializzazione, crea delle variabili speciali (chiamate variabili superglobali) che contengono diversi tipi di informazione. Queste variabili sono accessibili da qualunque posizione dello script ed hanno dei nomi riservati che non andrebbero sovrascritti se non in casi particolari. Le variabili in questione sono:

- `$_GET`: un array contenente tutti i parametri passati alla pagina tramite metodo GET (accodando all'URL, dopo un punto di domanda (?) una serie di assegnazioni di valori separate da &);
- `$_POST`: un array contenente tutti i parametri passati alla pagina tramite il metodo POST (solitamente attraverso un Form oppure attraverso delle chiamate create manualmente);
- `$_COOKIE`: un array contenente tutti i cookie validi nella pagina corrente con il rispettivo valore;
- `$_REQUEST`: un array che contiene i valori delle variabili precedenti, tutti insieme. In caso di omonimia delle variabili, queste sono soprascritte dando precedenza in base al valore della direttiva `variables_order` impostata nel file di configurazione di PHP (solitamente `php.ini`);
- `$GLOBALS`: un array contenente tutte le variabili che risultano globali nello scope corrente;
- `$_SERVER`: un array contenente delle variabili impostate dal Web Server oppure

direttamente legate all'ambiente di esecuzione dello script corrente (come ad esempio il browser dell'utente o il suo indirizzo IP). Vi sono molte variabili associate a questo array, pienamente descritte nella documentazione ufficiale di PHP. Le più utili sono:

- `PHP_SELF`: il nome del file dello script correntemente in esecuzione;
- `DOCUMENT_ROOT`: la root da cui viene eseguito lo script corrente;
- `REMOTE_ADDR`: l'indirizzo IP dell'utente che sta eseguendo lo script. Questo valore viene impostato in base ad un header impostato dal browser, quindi non andrebbe utilizzato come discriminante in situazioni di sicurezza critiche;
- `$_FILES`: un array contenente informazioni sui file inviati alla pagina tramite POST. Ad ogni chiave dell'array corrisponde un altro array contenente i dettagli sul file. Questi dettagli sono:
 - `name`: il nome del file caricato;
 - `tmp_name`: il path della cache temporanea del file;
 - `size`: le dimensioni in byte del file;
 - `type`: il mime type del file, se recuperabile;
 - `error`: un numero indicante lo stato dell'upload del file, che può essere utilizzato per controllare che l'upload sia avvenuto correttamente;
- `$_ENV`: un array contenente tutte le variabili d'ambiente accessibili da PHP;
- `$_SESSION`: un array contenente tutte le variabili di sessione accessibili dalla pagina corrente;

A fronte di tutto questo è importante ricordarsi di non sovrascrivere questi valori, dato che sono spesso fondamentali per la corretta esecuzione dello script.

Le costanti

Le costanti sono dei contenitori immutabili per dei valori semplici (stringhe e numeri), che possono essere accedute attraverso il loro nome senza che questi sia preceduto dal classico simbolo del dollaro (\$).

Le costanti in PHP possono essere definite manualmente oppure essere impostate automaticamente da PHP in base al contesto ed alle librerie caricate. Le costanti vengono impostate manualmente usando l'istruzione `define()`:

```
define('MIA_COSTANTE', 1);
define('SECONDA_COSTANTE', prova);
```

È convenzione specificare dei nomi composti solamente da caratteri maiuscoli o underscore. Le costanti possono essere accedute in questo modo:

```
echo MIA_COSTANTE;
echo "<br />";
echo SECONDA_COSTANTE;
```

Per controllare che una costante sia definita effettivamente è necessario utilizzare la funzione `defined`, che accetta come argomento la stringa che identifica il nome della costante da controllare.

```
/*
Il codice che segue avrà un comportamento differente da quello che ci aspettiamo.
NON_DEFINITA viene trasformata in una stringa NON_DEFINITA, viene restituito
un notice da PHP e l'espressione viene valutata vera, eseguendo quindi il codice
tra graffe che invece vorremmo saltare.
*/
if(NON_DEFINITA)
{
    // ....
}
```

```
// Questo è corretto
if(define('NON_DEFINITA'))
{
    // ...
}
```

PHP definisce automaticamente moltissime costanti, molte delle quali specifiche per le librerie importate. Le più importanti, indipendenti dalle librerie, e che spesso risultano utili sono le seguenti:

- `__FILE__`: il path del file in cui ci troviamo. In caso il file sia incluso da un altro in esecuzione `__FILE__` avrà comunque il nome del file incluso;
- `__CLASS__`: il nome della classe in cui ci troviamo attualmente;
- `__FUNCTION__`: il nome della funzione in esecuzione;
- `__METHOD__`: il nome del metodo in esecuzione;
- `__LINE__`: il numero di linea corrente;

I tipi di dato

Il PHP supporta diversi tipi di dati, che non devono essere impostate dal programmatore ma sono automaticamente assunte dal motore a meno che il programmatore stesso ne forzi il tipo attraverso apposite funzioni quali `settype` (pratica comunque sconsigliata, conviene modificare il valore piuttosto che il tipo di dato di una variabile). I dati possono essere:

- Integer
- Float
- String
- Array
- Object
- Resource

Vediamoli uno ad uno

Integer

Gli Integer, o interi, possono assumere diversi valori numerici esprimibili in differenti notazioni.

```
$a = 18; # decimale
$a = -18; # decimale negativo
$a = 022; # notazione ottale; equivalente a 18 decimale
$a = 0x12; # notazione esadecimale, equivalente a 18 decimale
```

Probabilmente, almeno per iniziare, utilizzerete soprattutto i numeri decimali, ma sappiate che il linguaggio accetta anche altre notazioni rispetto a quella decimale.

Float

Questo tipo di dati sono semplicemente i numeri in virgola mobile, ad esempio 9.876; la sintassi per utilizzarli è anche qui alquanto semplice:

```
$a = 9.876;
$a = 9.87e6;
```

Come vedete PHP accetta anche la notazione esponenziale.

Strings

Sulle stringhe c'è molto più da dire rispetto ai tipi di dati precedenti. La sintassi di base per le stringhe è:

```
$string = "Io sono una stringa";
```

Se vengono utilizzate le virgolette (""), il contenuto della stringa viene espanso (o, tecnicamente, "interpolato"), come nell'esempio successivo:

```
$num = 10;  
$string = "Il numero è $num";
```

che visualizzerà "Il numero è 10". Come in tutti i linguaggi, comunque, anche con il PHP ci sono i caratteri speciali che vanno fatti precedere da un simbolo di escape; ad esempio, provate il seguente esempio:

```
$num = 10;  
$string = "Il numero è \"$num\"";
```

Chi pensa che l'output di tale codice sia "Il numero è "10" si sbaglia: a parte il fatto che, così come è scritto, lo script darebbe un errore di compilazione, le virgolette sono caratteri speciali, ma non per questo non è permesso utilizzarle; la sintassi corretta per il comando riportato sopra è:

```
$num = 10;  
$string = "Il numero è \"\$num\"";
```

Altri caratteri speciali sono:

- \n -> newline
- \r -> carriage return
- \t -> tabulazione
- \\ -> backslash
- \\$ -> simbolo del dollaro

L'alternativa ai caratteri di escape, quando non ci siano contenuti da espandere, sono gli apici ('); ad esempio:

```
$string = '$ è il simbolo del dollaro';
```

visualizzerà proprio ciò che è contenuto fra gli apici. Attenzione a non cadere nel più consueto degli errori:

```
$num = 10;  
$string = 'Il numero è $num';
```

che non visualizzerà "Il numero è 10" bensì "Il numero è \$num". Quindi possiamo affermare che, con gli apici, il contenuto della stringa viene riportato letteralmente, ossia com'è effettivamente scritto al suo interno.

Una sintassi alternativa per la definizione delle stringhe ormai quasi caduta in disuso è la sintassi HEREDOC:

```
$string = <<<EOT  
Questa è una stringa  
multilinea  
e fatta utilizzando la sintassi  
HEREDOC  
EOT;
```

Questa sintassi presuppone l'utilizzo di un separatore di inizio e fine stringa. Preceduto da <<< e chiuso utilizzando il punto e virgola. Può essere utilizzato un separatore qualsiasi formato da caratteri, ma è convenzione utilizzare EOT.

Array

Il PHP supporta sia gli array scalari sia gli array associativi. In PHP, un array di valori può essere esplicitamente creato definendone gli elementi oppure la sua creazione può avvenire inserendo valori all'interno dell'array, ad esempio:

```
$a = array ("abc", "def", "ghi");
```

crea l'array definendo esplicitamente gli elementi dell'array, al contrario dell'esempio che segue:

```
$a[0] = "abc";  
$a[1] = "def";  
$a[2] = "ghi";
```

In questo caso, l'array viene creato con tre elementi; ricordiamo che il primo elemento di un array viene identificato dal numero "0": se ad esempio la lunghezza di un array è "5", esso conterrà sei elementi; l'elemento contrassegnato dall'indice "0", infatti, è il primo dell'array. Se invece, per aggiungere elementi ad un array (supponiamo che sia quello precedentemente creato) si utilizzano le parentesi quadre vuote, i dati vengono accodati all'array; ad esempio:

```
$a[] = "lmn";  
$a[] = "opq";
```

In questo caso, l'array si allunga di 2 elementi e risulta:

```
$a[0] = "abc";  
$a[1] = "def";  
$a[2] = "ghi";  
$a[3] = "lmn";  
$a[4] = "opq";
```

Questo esempio è molto utile quando si vogliono accodare degli elementi all'array senza ricorrere a specifiche funzioni e senza dover andare a leggere il numero di elementi contenuti nell'array: tutto sarà accodato automaticamente e correttamente.

Gli array associativi si basano invece su coppie "name-value"; un esempio potrebbe essere:

```
$a = array(  
"nome" => "Mario",  
"cognome" => "Rossi",  
"email" => "mario@rossi.com",  
);
```

È interessante la possibilità della funziona array di annidare le entries, come nell'esempio che segue:

```
$a = array(  
    "primo" => array(  
        "nome" => "Mario",  
        "cognome" => "Rossi",  
        "email" => "mario@rossi.com",  
    ),  
    "secondo" => array(  
        "nome" => "Marco",  
        "cognome" => "Verdi",  
        "email" => "mario@verdi.com",  
    )  
);
```

Eeguire su questo array un comando del tipo:

```
<? echo $a["secondo"]["email"]; ?>
```

visualizzerà "mario@verdi.com".

Objects

In PHP si possono utilizzare anche gli oggetti. Dato che PHP 5 ha ampliato enormemente il supporto alla programmazione ad oggetti questa verrà trattata successivamente in modo approfondito. Vediamo comunque un esempio:

```
class visualizza
{
    public function esegui_visualizza () {
        echo "Visualizza un messaggio";
    }
}
$obj = new visualizza();
$obj->esegui_visualizza();
```

Iniziamo definendo la classe "visualizza", che contiene la funzione "esegui_visualizza" che non fa altro che visualizzare un semplice messaggio a video. Con lo statement "new" inizializziamo l'oggetto "\$obj" e richiamiamo la funzione visualizza con l'operatore -> su \$obj. Più dettagli, come detto, in seguito.

Operatori di base

Gli operatori utilizzabili con PHP sono simili a quelli utilizzati con gli altri linguaggi di programmazione; per comodità, li divideremo in differenti "famiglie": gli operatori aritmetici, gli operatori di assegnazione a, in generale, tutti gli altri operatori.

Gli operatori aritmetici

Gli operatori aritmetici sono i più semplici, e sono:

- Addizione \$a + \$b
- Sottrazione \$a - \$b
- Moltiplicazione \$a * \$b
- Divisione \$a / \$b
- Resto della divisione \$a % \$b

Fermiamoci per un attimo sugli ultimi due per notare come il risultato della divisione non è approssimato all'intero più vicino, ma riporta tutto il numero risultante; il numero dei caratteri dopo il punto da considerare è definito nel file php.ini alla riga:

```
precision = 14
```

Quindi, verranno riportati "solo" 14 numeri dopo la virgola, a patto che ci siano. Nell'esempio:

```
$a = 12;
$b = 5;
$c = $a / $b;
echo $c;
```

il risultato è 2.4 e verrà visualizzato proprio come 2.4, non come 2.40000000000000. Quindi, i 14 decimali vengono visualizzati solamente se esistono, e non indiscriminatamente come inutili zeri. Il resto della divisione viene riportato da solo, senza il risultato della divisione stessa: se nell'esempio precedente avessimo utilizzato "%" al posto di "/", il risultato sarebbe stato "2".

Assegnazione

Spesso, purtroppo, gli operatori di assegnazione vengono confusi con l'operatore di uguaglianza; l'operatore "=" ha un suo significato, che non va confuso con quello di "==". L'operatore di assegnazione è il simbolo dell'uguale (=) che attribuisce ad una variabile un valore; ad esempio

```
$a = 2;
```

imposta per "\$a" il valore "2". L'errore che si fa più spesso è scrivere qualcosa del tipo:

```
if ($a=2) {  
    // istruzioni;  
}
```

che, letto da occhi inesperti, potrebbe sembrare un'espressione per affermare che, se \$a è UGUALE a 2 deve venire eseguito il codice fra parentesi graffe. Ebbene, non è assolutamente così: se avessimo voluto scrivere ciò che è appena stato detto, avremo dovuto utilizzare:

```
if ($a == 2) {  
    // istruzioni;  
}
```

Altri operatori

Gli operatori che il PHP ci mette a disposizione sono molti, e vedremo in questa pagina i più importanti che non abbiamo ancora avuto modo di esaminare, in particolari gli operatori booleani, quelli matematici e quelli di incremento-decremento. Ad ogni operatore faremo seguire una breve descrizione.

`$a & $b`

operatore "And" (\$a e \$b);

`$a && $b`

come sopra, ma con una precedenza più alta;

`$a | $b`

operatore "Or" (\$a oppure \$b);

`$a || $b`

come sopra, ma con una precedenza più alta;

`$a ^ $b`

operatore "Xor" (\$a oppure \$b ma non entrambi);

`!$a`

operatore "Not" (vero se \$a non è vera);

`$a == $b`

operatore di uguaglianza, vero se \$a ha lo stesso valore di \$b;

`$a != $b`

l'opposto di quanto sopra;

`$a < $b;`

\$a è minore di \$b;

`$a <= $b`

\$a è minore o uguale a \$b;

`$a > $b`

\$a è maggiore di \$b;

`$a >= $b`

\$a è maggiore o uguale a \$b;

`$a ? $b : $c`

questo, da utilizzarsi più con le espressioni che con le variabili, valuta \$b se \$a è vera, e valuta \$c se \$a è falsa;

`++$a`

incrementa di uno \$a e la restituisce; `$a++` restituisce \$a e la incrementa di uno

--\$a

\$a--

come i due precedenti, ma il valore è decrementato.

Strutture di controllo

Non possono mancare in un linguaggio di programmazione le strutture di controllo, che permettono al programmatore di far compiere delle azioni al programma nel caso si verifichino (o non si verifichino) determinate condizioni.

If

If permette di eseguire un blocco di codice se avviene (o non avviene) una determinata condizione; la sua sintassi è:

if (condizione) statement

Ad esempio, vogliamo che uno script ci indichi se due variabili sono uguali:

```
$a = 2;
$b = 2;
if ($a == $b) {
    echo "$a è uguale a $b e valgono $a.\n";
}
```

If può anche essere utilizzato in maniera differente da quella appena esposta: eccone un esempio:

```
<? $a = 2; $b = 2; if ($a == $b) : ?>
    $a è uguale a $b.
<? endif; ?>
```

il cui operato è identico a quello esposto sopra anche se molto meno leggibile.

If può essere utilizzato anche senza le parentesi graffe, utilizzando endif quando si intende terminare il blocco "if"; ad esempio:

```
if ($a == $b)
    echo "$a è uguale a $b e valgono $a.\n";
endif;
```

Else

Else viene in "completamento" di if: con if, infatti, stabiliamo che succeda qualcosa all'avverarsi di una condizione; con else possiamo stabilire cosa accade nel caso questa non si avveri. Un esempio potrebbe essere:

```
$a = 2;
$b = 3;
if ($a == $b) {
    echo "$a è uguale a $b e valgono $a.\n";
} else {
    echo "$a è diversa da $b.\n$a vale \"$a\" mentre $b vale \"$b\".\n";
}
```

Elseif

Elseif permette di specificare casualità non definite da if; un esempio potrebbe essere: "Se \$a è uguale a \$b visualizza \$a, se \$a è diversa da \$b visualizza un messaggio d'errore, avvisa se \$a non esiste, avvisa se \$b non esiste". Con i soli if ed else non si potrebbe fare, ma con elseif diventa

semplice:

```
if ($a == $b) {
    echo "\"$a è uguale a \"$b.\"n\"";
} elseif ($a != $b) {
    echo "\"$a è diversa da \"$b.\"n\"";
} elseif (!$a) {
    echo "\"$a non esiste.\"n\"";
} elseif (!$b) {
    echo "\"$b non esiste.\"n\"";
}
```

Notate due cose: possono esserci, in un blocco, tutti gli elseif di cui avete bisogno e, per chi conosca il Perl, attenzione a non scrivere elsif al posto di elseif: il significato è lo stesso ma "elsif" non viene riconosciuto dal PHP così come elseif non viene riconosciuto dal Perl!

While

La condizione while si comporta esattamente come in C; la sintassi di base è:

```
while (espressione) statement
```

Come if, inoltre, while può essere utilizzato con o senza parentesi graffe, aggiungendo nel secondo caso lo statement endwhile. I due esempi che seguono si equivalgono:

```
/* Primo esempio: */
/* $a viene incrementato e visualizzato */
/* finchè il suo valore non supera "5" */
$a = 1;
while ($a <= 5) {
    print $a++;
}

/* Secondo esempio */

$a = 1;
while ($a <= 5)
    print $a++;
endwhile;
```

Tradotte, queste espressioni fanno in modo che, finchè (while) \$a è minore o uguale a "5", \$a viene incrementato di un'unità e visualizzato.

For

Anche for si comporta esattamente come avviene in C o in Perl; dopo il for, devono essere inserite tre espressioni che, finchè restituiscono "TRUE" permettono l'esecuzione dello statement che segue: considerate questo esempio:

```
for ($a = 0 ; $a <=10 ; $a++) {
    print $a;
}
```

che visualizzerà i numeri da "0" a "10". Nelle tre espressioni fra parentesi abbiamo definito che:

\$a ha valore 0;

\$a è minore o uguale a 10;

\$a è incrementata di una unità;

Quindi, per ogni valore di \$a a partire da "0" fino a "10" \$a viene visualizzato. È possibile omettere alcune operazioni (ricordandosi comunque di specificare sempre il punto e virgola come separatore)

nel caso in cui l'inizializzazione, il controllo o la post esecuzione siano effettuate in altri luoghi oppure non debbano essere effettuate.

Switch

Switch permette di sostituire una serie di if sulla stessa espressione e, ovviamente, di agire dipendentemente dal valore di questa:

```
switch ($i) {
    case 0:
        echo "\$i vale 0";
        break;
    case 1:
        echo "\$i vale 1";
        break;
}
```

Abbiamo qui introdotto l'istruzione break che permette di uscire da un blocco nel caso si avveri una determinata condizione.

Foreach

PHP 4 (non PHP 3) permette l'uso della struttura di controllo foreach, alla stessa maniera del linguaggio Perl e altri. Ciò semplicemente fornisce una facile metodo per attraversare un array. foreach funziona solo con le matrici e genera un errore se si tenta di utilizzarlo con variabili di tipo differente oppure non inizializzate. Esistono due possibili notazioni sintattiche; la seconda è un'utile estensione della prima:

```
foreach(array_expression as $value)
    istruzione
foreach(array_expression as $key => $value)
    istruzione
```

La prima attraversa l'array dato da array_expression. Ad ogni ciclo, si assegna il valore dell'elemento corrente a \$value e il puntatore interno avanza di una posizione (in modo tale che al ciclo successivo l'elemento corrente sarà il successivo elemento dell'array).

La seconda esegue lo stesso ciclo con la differenza che il valore dell'indice corrente viene assegnato ad ogni ciclo, alla variabile \$key. Sono anche possibili cicli sugli oggetti.

Nota: All'inizio dell'esecuzione di un ciclo foreach il puntatore interno viene automaticamente posizionato nella prima posizione. Questo significa che non è necessario utilizzare la funzione reset() prima di un ciclo foreach.

Nota: A meno che la matrice non sia per riferimento, foreach agisce su una copia e non sulla matrice stessa. Pertanto il puntatore dell'array originale non viene modificato come accade utilizzando la funzione each() e le modifiche agli elementi dell'array non appaiono nell'array originale. Tuttavia il puntatore interno della matrice originale viene avanzato durante l'elaborazione della matrice. Se si assume che il ciclo foreach giunga al termine, allora si avrà che il puntatore interno della matrice sarà al termine della matrice stessa.

Dal PHP 5 si possono modificare facilmente gli elementi di una matrice antepoendo & a \$value. Questo assegna un riferimento anzichè copiare il valore.

```
<?php
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
    $value = $value * 2;
```

```
}  
// $arr vale ora array(2, 4, 6, 8)  
?>
```

Questo è possibile soltanto se l'array indicato può essere referenziato (ad esempio è una variabile).

Nota: foreach non offre la possibilità di annullare la generazione di messaggi d'errore utilizzando il carattere '@'.

I due cicli seguenti sono identici da un punto di vista funzionale:

```
<?php  
$arr = array("one", "two", "three");  
reset ($arr);  
while (list(, $value) = each ($arr)) {  
    echo "Valore: $value<br />\n";  
}  
  
foreach ($arr as $value) {  
    echo "Valore: $value<br />\n";  
}  
?>
```

Le funzioni built-in

Come nella guida precedente, i prossimi paragrafi analizzeranno alcune delle funzioni fornite nativamente da PHP. Le funzioni native di PHP sono talmente tante che necessiteremmo di centinaia di pagine solamente per trattarle in modo superficiale. Per questo motivo ho deciso di dare una rapida occhiata ad alcune di queste, lasciandovi il compito di seguire la guida ufficiale per i dettagli o per lo studio delle altre funzioni.

Le funzioni sono una delle parti più importanti di un linguaggio di programmazione, perchè permettono di eseguire determinate operazioni all'interno di uno script.

Le funzioni messe a disposizione dal PHP sono moltissime, e vederle tutte sarebbe inutile; ci soffermeremo invece su quelle più importanti ordinate alfabeticamente.

abs: restituisce il valore assoluto di un numero:

```
$num = -3.4;  
$a = abs($a);  
echo $a, "\n";
```

restituirà 3.4

acos: restituisce l'arcocoseno dell'argomento:

```
$arg = 1;  
$arc_cos = acos($arg);  
echo "$arc_cos\n";
```

restituirà 0.

array: si veda la precedente spiegazione riguardo i tipi di dati;

asin: restituisce il seno dell'argomento;

atan: restituisce l'arcotangente dell'argomento;

base64_decode: decodifica una stringa codificata in MIME base64 (vedi sotto);

base64_encode: codifica dati in MIME base64; ad esempio con:

```
$str = "Ciao, io sono pippo\n";  
echo "$str\n";  
$enc_str = base64_encode($str);
```

```
echo "$enc_str\n";
$dec_str = base64_decode($enc_str);
echo "$dec_str\n";
```

si passa allo script la stringa "\$str" che viene prima codificata e visualizzata, poi decodificata e nuovamente visualizzata;

basename: restituisce, dato un percorso, la componente di questo identificata da un nome di file; ad esempio:

```
$path = "/var/www/php/index.php";
$base = basename($path);
echo "$base\n";
```

restituirà "index.php";

bcadd: somma due numeri;

```
$num = bcadd(1.334, 4.44554, 2);
echo "$num\n";
```

restituirà 5.77; la funzione "bcadd" prende come primi due argomenti due numeri e, come terzo argomento opzionale, il numero di cifre da visualizzare dopo la virgola;

bccomp: compara due numeri: la funzione prende come argomento due numeri e, opzionalmente, un ulteriore numero che determina il numero di decimali da considerare dopo la virgola per considerare i due numeri uguali; restituisce "0" nel caso i due numeri siano uguali, "+1" se il numero di sinistra è maggiore di quello di destra e "-1" nel caso opposto. Considerate il seguente esempio:

```
$comp = bccomp(0.334, 0.301, 2);
echo $comp;
```

che restituirà "1"; ma se, al posto del "2" avessimo inserito uno oppure non avessimo inserito niente, il risultato sarebbe stato "0".

bcdiv: divide due numeri, con le stesse modalità descritte per "bcadd" e "bccomp";

bcmult: moltiplica due numeri, ed è possibile aggiungere un ulteriore parametro per limitare il numero di cifre dopo la virgola:

```
$molt = bcmul(2.31, 3.21, 2);
echo "$molt\n";
```

restituirà 7.41;

bcpow: eleva a potenza due numeri, con la possibilità di specificare il numero di cifre dopo la virgola:

```
$pot = bcpow(2.3, 3, 2);
echo "$pot\n";
```

eleverà 2.3 alla terza potenza, approssimando il risultato alla seconda cifra decimale;

bcsqrt: calcola la radice quadrata di un numero, con la possibilità di approssimare il numero di cifre dopo la virgola aggiungendo un secondo elemento alla funzione (come avveniva per altre funzioni matematiche viste sopra);

bcsub: sottrae un numero da un altro, anche qui con la possibilità di approssimare le cifre dopo la virgola:

```
$num = bcsub(2, 5);
echo "$num\n";
```

restituirà "-3";

bin2hex: converte una stringa di dati dal formato binario a formato esadecimale;

ceil: restituisce il valore intero più alto riferito al numero passato come argomento alla funzione:

```
$num = ceil(3.22112);  
echo "$num\n";
```

restituisce "4";

chdir: cambia la directory di lavoro:

```
$dir = "/var/www/";  
chdir($dir);
```

restituisce TRUE se l'operazione ha successo, FALSO in caso contrario, ad esempio nel caso la directory non sia leggibile;

checkdate: controlla che una data sia valida; per considerarsi valida, una data deve avere:

- l'anno compreso fra "0" e "32767";
- il mese compreso fra "1" e "12";
- il giorno è compreso fra "0" ed il numero relativo al numero di giorni del mese a cui si fa riferimento;

chgrp: tenta di cambiare il gruppo di un file a "gruppo"; la funzione accetta come argomenti il nome del file a cui si vogliono cambiare i permessi ed il nuovo gruppo di appartenenza:

```
chgrp(filename, gruppo);
```

Nei sistemi Windows non funziona ma restituisce sempre vero.

chmod: è equivalente al comando di sistema Unix "chmod" ed ha la stessa sintassi di chgrp;

chomp: rimuove i "whitespaces" da una stringa; spesso è utilizzato per eliminare i caratteri "\n" quando si riceve un argomento dallo standard input; il carattere eliminato può essere letto con:

```
$carattere = chop($string);  
echo "$carattere\n";
```

chown: cambia il proprietario di un file, come l'analogo comando di sistema Unix. Accetta come argomento il nome del file ed il nome del nuovo proprietario:

```
$file = "prova.txt";  
chown($file, $user);
```

Nei sistemi Windows, non fa niente e restituisce sempre vero (ed è peraltro inutile inserire questa funzione all'interno di uno script che non supporta il comando "chown");

chr: restituisce il carattere ASCII specificato dal rispettivo numero; immagino sappiate, ad esempio, che la combinazione "Alt + 0126" restituisce la tilde (~); lo si può vedere con il seguente codice:

```
$ascii= "0126";  
$char = chr($ascii);  
echo "$char\n";
```

chunk_split: divide una stringa in parti di "n" caratteri; il numero è passabile alla funzione dopo la stringa da dividere. Se non impostato, di default è assunto come 76; l'esempio

```
$string = "Questo è un corso per imparare il linguaggio php";  
$split = chunk_split($string, 5);
```

restituirà:

```
Quest  
o è u  
n cor  
so pe  
r imp
```

```
arare  
il l  
ingua  
ggio  
php
```

La funzione è utile per l'encoding MIME base64 visto precedentemente con la funzione `base64_encode`;

closedir: chiude una directory precedentemente aperta con la funzione `opendir()` - vedi;

copy: crea la copia di un file:

```
$file = "prova.txt";  
copy($file, "$file.bak");
```

cos: restituisce il valore del coseno dell'argomento;

count: conta gli elementi in una variabile; ad esempio:

```
$arr[0] = "abc";  
$arr[1] = "def";  
$arr[2] = "ghi";  
$count = count($arr);  
echo $count;
```

restituirà "3", visto che all'interno dell'array "\$arr" sono presenti 3 elementi (\$arr[0], \$arr[1], \$arr[2]);

crypt: critta una stringa; la sintassi della funzione `crypt()` è:

```
crypt(string, salt);
```

In pratica, dovremo passare alla funzione la stringa che dovrà essere crittata e, opzionalmente, il seme con cui crittarla; se questo non è passato alla funzione, sarà generato in maniera random dal PHP stesso. Un esempio di crittazione di una stringa potrebbe essere il seguente:

```
$var = "Questa è una variabile";  
$crypt = crypt($var, "aa");  
echo $crypt;
```

che restituirà la stringa crittata;

current: restituisce il primo elemento di un array:

```
$arr[0] = "abc";  
$arr[1] = "def";  
$arr[2] = "ghi";  
$current = current($arr);  
echo $current;
```

visualizzerà "abc";

date: visualizza la data in formato che è possibile definire; la funzione riconosce come validi i seguenti formati:

- a: am/pm;
- A: AM/PM
- d: giorno del mese in due cifre, da "0" a "31";
- D: giorno del mese in formato testo, ad esempio "Mon";
- F: mese, in formato testuale, ad esempio "March";
- h: ora nel formato "01", "12";
- H: ora nel formato "00", "23";
- g: ora nel formato "1", "12";

- G: ora nel formato "0", "23";
- i: minuti, nel formato "00", "59";
- j: giorno del mese nel formato "1", "31";
- l: giorno della settimana, ad esempio "Monday";
- L: specifica se l'anno è bisestile o meno ("1" oppure "0");
- m: mese nel formato "01", "12";
- n: mese nel formato "1", "12";
- M: mese in formato testuale corto, ad esempio "Jan";
- s: secondi da "00" a "59";
- S: suffisso inglese per gli ordinali, "st", "nd", "rd", "th";
- t: numero di giorni nel mese corrente, da "28" a "31";
- w: giorno della settimana in formato numerico ("0"=domenica);
- Y: anno in quattro cifre, ad esempio "2000";
- y: anno in due cifre, ad esempio "00";

Ad esempio, si potrebbe scrivere:

```
echo (date("l d F y H:i:s a"));
```

per avere la data corrente, che ad esempio potrebbe essere:

```
Friday 23 June 00 11:45:48 am
```

debugger_off: disabilita il debugger PHP;

debugger_on: abilita il debugger PHP;

decbin: converte un numero da decimale a binario; ad esempio, il numero "10" decimale è "1010" in formato binario, e con del semplice codice PHP potremo scrivere:

```
$bin = decbin(10);
echo $bin, "\n";
```

che restituirà appunto "1010";

dechex: converte un numero da decimale a esadecimale; la sintassi è identica a quella utilizzata per decbin();

decoct: converte un numero da formato decimale a formato ottale; la sintassi è la stessa utilizzata per decbin();

define: definisce una costante; come abbiamo visto nel capitolo sulle costanti, queste sono simili alle variabili solamente che non hanno il simbolo del dollaro davanti; per definire una costante si utilizza la seguente sintassi:

```
define("COSTANTE", "Questa è una costante");
echo COSTANTE;
```

L'esempio riportato sopra visualizzerà "Questa è una costante";

defined: controlla che una certa costante esista: un esempio potrebbe essere:

```
define("COSTANTE", "Questa è una costante");
```

```
if (defined("COSTANTE")) {
    echo "La costante è definita\n";
} else {
    echo "La costante non è definita\n";
}
```

che visualizza un messaggio a seconda che la costante sia o meno definita;

die: visualizza un messaggio ed esce dal programma:

```
if (defined($num)) {
    echo "\$num è definito\n";
} else {
```

```
}
die ("\$num non è definito; impossibile proseguire\n");
}
```

dirname: quando si specifica un path, riporta il path senza il nome del file finale: se ad esempio il path è `"/home/yourname/public_html/index.php"` la funzione restituirà solamente `"/home/yourname/public_html"`:

```
$path = "/home/yourname/public_html";
echo(dirname($path));
```

Avrete notato che questa funzione fa l'esatto contrario della funzione `basename()`; combinando le due funzioni, si può avere il path completo di un file, compreso il suo nome;

diskfreespace: restituisce lo spazio di disco libero; se volessimo ad esempio vedere quanto spazio rimane nella directory root della macchina, potremo scrivere:

```
echo(diskfreespace("/"));
```

each: restituisce il primo valore di un array utilizzando le keys 0, 1, key e value; se l'array è associativo, si può scrivere:

```
$array = array ("nome" => "valore", "nome2" => "valore2");
while (list($key, $val) = each ($array)) {
    echo "$key => $val\n";
}
```

che restituirà:

```
nome => valore
nome2 => valore2
```

Se invece l'array non è associativo, il codice sarà:

```
$array = array ("nome", "valore", "nome2", "valore2");
while (list($key, $val) = each ($array)) {
    echo "$key => $val\n";
}
```

ed il risultato:

```
0 => nome
1 => valore
2 => nome2
3 => valore2
```

Come avrete certamente notato, la funzione `each()` viene spesso utilizzata insieme a `list()`, che vedremo in seguito;

echo: visualizza una o più stringhe; non penso ci sia molto da dire su questa funzione, vista sia la sua semplicità sia le numerose implicazioni in cui l'abbiamo vista in uso.

ereg_replace: sostituisce un'espressione regolare con determinati valori; alla funzione devono essere passati tre argomenti: il primo indica il testo da sostituire, il secondo è il testo utilizzato per la sostituzione ed il terzo è la stringa da modificare; ad esempio:

```
$stringa = "Questo è un corso su ASP";
echo ereg_replace("ASP", "PHP", $stringa);
```

Notate che si sarebbe potuto scrivere anche:

```
echo ereg_replace("ASP", "PHP", "Questo è un corso su ASP");
```

che non avrebbe avuto molto senso, comunque.

ereg: esegue il matching di un'espressione regolare. L'esempio fornito con la documentazione è alquanto eloquente:

```
if (ereg("[0-9]{4})-([0-9]{1,2})-([0-9]{1,2})", $date, $regs) {
    echo "$regs[3].$regs[2].$regs[1]";
} else {
    echo "Invalid date format: $date";
}
```

Tutto questo ciclo è fatto per controllare che una data sia in formato corretto. Vediamo il significato di "[0-9]{4})-([0-9]{1,2})-([0-9]{1,2})". Per chi conosca le espressioni regolari, non sarà difficile tradurre quanto sopra con "un numero da 0 a 9 ripetuto quattro volte seguito da un '-', da un numero da 0 a 9 ripetuto una o due volte, da un '-' e da un numero da 0 a 9 ripetuto una o due volte". Come spesso accade, leggere un'espressione regolare è molto più semplice che tradurla nel linguaggio parlato.

ereg_replace: funziona esattamente come `ereg_replace()`, solamente che in questo caso l'espressione regolare è sostituita in maniera "case insensitive", ossia ignorando i caratteri maiuscoli e minuscoli;

eregi: funziona esattamente come `ereg()`, solamente che in questo caso l'espressione regolare è sostituita in maniera "case insensitive";

error_log: invia un messaggio di errore al file di log del Web Server, direttamente alla porta TCP dalla quale è arrivata la richiesta o in un file. La sintassi è:

```
error_log(message, message_type, destination);
```

Message_type è un numero che specifica dove deve arrivare il messaggio. Può essere:

- 0: il messaggio è inviato al logger del PHP o nel file specificato da "error_log";
- 1: il messaggio è inviato per email al parametro (probabilmente un valido indirizzo email) specificato in "destination"; 2: il messaggio è inviato al debugger;
- 3: il messaggio è inserito in append al parametro specificato in "destination";

escapeshellcmd: se si richiama un comando esterno da una shell, con questo comando si fa in modo che i metacaratteri della shell vengano fatti precedere da un carattere di escape per evitare che il comando produca degli errori;

exec: esegue un programma esterno;

exit: esce da uno script; il comando `exit()` è utile nei casi si voglia fermare uno script in caso qualcosa non soddisfi determinate condizioni, ad esempio:

```
if (condizione) {
    esegui il blocco;
} else {
    exit;
}
```

Ricordate che `exit()` non riporta un messaggio di errore come fa `die()`: se vi interessa dare "spiegazioni" sul perchè lo script termina, utilizzate `die()`, ma ricordate che non è possibile scrivere:

```
exit "Esco dal programma\n";
```

o meglio, è possibile ma non ha alcun effetto se non quello di uscire;

exp: eleva "e" (2.71828.....) alla potenza riportata come argomento della funzione:

```
echo exp(3);
```

restituirà: 20.0855369...

explode: divide una stringa secondo un determinato pattern. Ad esempio, volendo dividere una stringa contenente tre nomi separati da virgole possiamo scrivere:

```
$nomi = "Tizio,Caio,Sempronio";  
list ($nome1, $nome2, $nome3) = explode(",", $nomi);  
echo "$nome1\n$nome2\n$nome3\n";
```

che restituirà:

```
Tizio  
Caio  
Sempronio
```

Explode() è una versione semplificata di split(), che vedremo in seguito. Entrambe le funzioni, inoltre, sono molto utili nel caso ci sia la necessità di leggere determinati file contenenti delle liste;
fclose: chiude un puntatore ad un file precedentemente aperto con fopen(). Si veda fopen() per maggiori informazioni;

feof: testa un puntatore ad un file per vedere se si è alla fine dello stesso;

fgetc: restituisce il primo carattere del puntatore precedentemente aperto con fopen(); se ad esempio il puntatore \$file punta al file "/tmp/prova.txt" che contiene solamente la riga "Ciao", un codice come il seguente:

```
$char = fgetc($file);  
echo "$char\n";
```

restituirà "C" (ovviamente senza virgolette);

file_exists: controlla se un file esiste, riportando TRUE in caso positivo o FALSE in caso negativo; ad esempio:

```
if (file_exists($file)) {  
print "$file esiste";  
}
```

Può essere molto utile utilizzare questa funzione nel caso sia necessario agire su uno o più file, in modo da agire sullo stesso solo nel caso questo esista senza rischiare di incorrere in inspiegabili "anomalie" dello script;

filegroup: restituisce il gruppo al quale appartiene il file:

```
$filename = "/tmp/prova.txt";  
$group = filegroup($filename);  
echo "$filename appartiene al gruppo $group\n";
```

Ovviamente, la funzione è implementata nei soli sistemi multiuser;

filesize: restituisce la grandezza di un file:

```
$filename = "/tmp/ptova.txt";  
$size = filesize($filename);  
echo "$filename -> $size\n";
```

filetype: determina il tipo di file; i valori possibili sono: fifo, char, dir, block, link, file e unknown;

flock: applica il locking ad un file; specificamente, flock() opera su un puntatore ad un file precedentemente aperto e le operazioni possibili sono:

- 1: per il lock in lettura;
- 2: per il lock in scrittura;
- 3: per rimuovere il lock, di qualsiasi tipo sia;
- 4: per impedire che flock() blocchi un file mentre applica il lock;

Ad esempio, per applicare flock() ad un puntatore "\$file" precedentemente definito occorrerà scrivere:

```
/* Per impedire che il file sia letto*/  
flock($file, 2);  
/* Codice per lavorare sul file */
```

```
.....  
/* Per rimuovere il flock */  
flock($file, 3);
```

fopen: apre un file oppure un'URL. La sintassi è `fopen(filename, mode)`; Ovviamente a "filename" corrisponde il nome del file o l'URL dello stesso, ed a "mode" la modalità con il quale questo deve essere aperto: si ha qui la possibilità di scegliere fra:

- r: apre il file in sola lettura;
- r+: apre il file in lettura ed in scrittura;
- w: apre il file in sola scrittura;
- w+: apre il file in lettura e scrittura;
- a: apre il file in sola scrittura e inserisce il puntatore alla fine del file ("w" lo inserisce alla fine)
- a+: apre il file in lettura e scrittura inserendo il puntatore alla fine del file;

Ad esempio, per aprire un file locale in sola lettura scriveremo:

```
$file = fopen("/tmp/prova.txt", "r");
```

Per un URL, invece:

```
$file = fopen("http://www.myhost.com/index.html", "r");
```

Per tutte le successive operazioni sul file, poi, dovremo agire direttamente sul puntatore (\$file) e non direttamente sul file;

header: invia un qualsiasi header HTTP; ad esempio:

```
header("Pragma: no-cache");
```

Questa funzione è molto utile in diversi casi: ad esempio, per forzare un'autorizzazione, per inviare un errore "301" o via dicendo;

hexdec: restituisce il valore decimale di un numero esadecimale;

implode: come risulta dal nome, questa funzione non è che l'opposto di `explode`: la sintassi è identica, ma in questo caso restituisce una stringa con i valori separati dal primo argomento della funzione;

in_array: restituisce valore vero se in un array è presente un determinato valore; un esempio potrebbe essere:

```
$numeri = array("1", "2", "3", "4", "5");  
$num = 2;  
if (in_array($num, $numeri)) {  
    print "$num è presente nell'array \ $numeri\n";  
}
```

is_array: controlla se una data variabile è un array:

```
if (is_array($var)) {  
    echo "\ $var è un array\n";  
}
```

La stessa cosa viene fatta, con ovviamente la differenza dell'argomento, dalle funzioni:

- is_dir;
- is_double;
- is_executable;
- is_file;
- is_float;
- is_int;
- is_integer;

- `is_link`;
- `is_long`;
- `is_object`;
- `is_readable`;
- `is_real`;
- `is_string`;
- `is_writeable`.

isset: restituisce TRUE nel caso la variabile esista, falso nel caso opposto; ad esempio, per vedere se esiste o meno una variabile, è possibile scrivere:

```
$a = 10;
echo isset($a), "\n";
echo isset($b), "\n";
```

che restituirà 1 e 0; ricordiamo che 1 è considerato valore di successo (TRUE), 0 di insuccesso (FALSE);

join: unisce gli elementi di un array con una determinata stringa; l'uso è identico a quello di `implode()`;

key: prende una chiave da un array associativo; un semplice esempio potrebbe essere:

```
$array = array("1" => "uno");
$chiave = key($array);
echo "$chiave\n";
```

che restituirà "1". Questa funzione è utile per estrarre tutte le chiavi di array associativi complessi;

link: crea un hard link; la sintassi è `link(target, link)`;

Le informazioni sui link (a proposito dell'esistenza del file a cui punta il link stesso) possono essere visualizzate con `linkinfo()`;

list: assegna delle variabili come se fossero parti di un array; riprendiamo l'esempio fatto con `each`:

```
$array = array ("nome" => "valore", "nome2" => "valore2");
while (list($key, $val) = each ($array)) {
    echo "$key => $val\n";
}
```

In questo caso, `list()` è utilizzato per "stilare" una lista di variabili che verranno estratte dall'array, senza ovviamente dare loro un valore ma lasciando alle parti successive del codice l'assegnazione dei loro valori. È inoltre utile notare che le variabili create da `list()` non assumono un solo valore, ma per ogni chiamata assumono un diverso valore, a seconda degli elementi presenti nell'array.

mail: funzione per l'invio di email; la funzione ha sintassi `mail(To, Subject, Message, Altri_headers)`;

Supponendo di voler inviare un'email a "nome@host.com" con subject "Prova" e volendo specificare il nome del mittente, possiamo scrivere:

```
mail("nome@host.com", "Subject", "Questo è il corpo dell'email", "From: mittente <mittente@host.net>);
```

Come vedete, inviare delle email tramite script PHP ed utilizzando la funzione "mail" è molto semplice. Ovviamente, nel file di configurazione, dovrete aver specificato la locazione di `sendmail` (o analogo programma per l'invio delle email);

max: restituisce il valore più alto di una serie di variabili, ad esempio:

```
$num = 1;
$num2 = 23;
$num3 = 0.3;
$max = max($num, $num2, $num3);
echo $max, "\n";
```

restituirà "23". Opposto a max() è min(), che adotta la stessa sintassi di max());

mkdir: crea una directory, di cui si deve specificare il percorso ed i permessi:

```
mkdir("/tmp/prova", 0777);
```

creerà la directory "/tmp/prova" con permessi impostati a 0777;

opendir: apre una directory, della quale sarà possibile leggere gli elementi con readdir() e, successivamente, chiuderla con closedir();

phpinfo: è la funzione più "rappresentativa" del PHP, in quanto visualizza moltissime informazioni sul PHP stesso: l'uso dovrebbe essere noto:

```
phpinfo();
```

phpversion: visualizza la versione di PHP che si sta utilizzando;

popen: apre un puntatore ad un processo che deve essere chiuso con pclose();

print: visualizza una stringa a video come echo();

rand: genera un valore numerico in maniera casuale; se si volesse un valore compreso fra 10 e 20, si potrebbe scrivere:

```
$random = rand(10, 20);
```

range: crea un array contenente un range di valori interi specificato; ad esempio, per creare un array con valori da 1 a 10 sarà necessario scrivere:

```
$array = range(1, 10);
```

rename: rinomina un file: ad esempio, si usa

```
rename("oldname", "newname");
```

per rinominare "oldname" come "newname";

rmdir: come l'analogo somando unix, rimuove una directory; questo può essere fatto solo se:

- la directory è vuota;
- i permessi sulla directory lo consentono.

round: arrotonda un numero:

```
$numero = round(2,3); /* restituisce 2 */  
$numero = round(2.5); /* restituisce 3 */  
$numero = round(2.6); /* restituisce 3 */
```

Come avrete notato, i decimali da 0 a 4 sono approssimati all'intero precedente, da 5 a 9 all'intero successivo

shuffle: ordina in modo casuale gli elementi di un array; ad esempio, per poter visualizzare gli elementi di un array in maniera casuale si potrebbe scrivere:

```
$num = range(0,10);  
shuffle($num);  
while (list($numero) = each($num)) {  
    echo "$numero ";  
}
```

sin: restituisce il seno dell'espressione;

sizeof: calcola il numero di elementi presenti in un array. Se ad esempio si volesse calcolare il numero di elementi in un array ed agire di conseguenza, si potrebbe scrivere:

```
$array = array("1", "2", "3", "4", "5");  
$size = sizeof($array);  
if ($size <= 10) {  
    echo "L'array contiene meno di 10 elementi\n";  
}
```

```
} else {  
    echo "L'array contiene più di 10 elementi\n";  
}
```

sleep: mette lo script in pausa per un determinato numero di secondi, specificato come argomento della funzione; ad esempio, sleep(10) farà in modo che lo script venga sospeso per 10 secondi, per poi continuare normalmente;

split: divide una stringa a seconda di un determinato pattern; ad esempio:

```
$linea = "tizio|caio|sempronio";  
list ($uno, $due, $tre) = split("\\|", $linea, 3);  
print "1 => $uno\n2 => $due\n3 => $tre\n";
```

Notate che è stato necessario inserire un carattere di escape (\) prima di ogni "|" nell'espressione da utilizzare per dividere la riga;

sqrt: restituisce la radice quadrata dell'argomento;

strcmp: esegue una comparazione su due stringhe: ad esempio:

```
$cmp = strcmp("Ciao", "Ciao a tutti");  
if ($cmp == "0") {  
    print "Le stringhe sono identiche\n";  
} elseif ($cmp < 0) {  
    print "La seconda riga è più lunga della prima\n";  
} elseif ($cmp > 0) {  
    print "La prima riga è più lunga della prima\n";  
}
```

restituisce "La seconda riga è più lunga della prima". La funzione, infatti, restituisce "0" se le stringhe sono uguali, un valore minore di zero se la seconda è più lunga della prima e maggiore di zero se la prima è più lunga della seconda;

system: esegue un programma di sistema, ne restituisce l'output e ritorna allo script;

tan: restituisce la tangente dell'argomento;

unset: elimina il valore di una variabile;

usleep: come sleep(), ma questa funziona blocca lo script per N microsecondi.

Crittazione

Con questo, abbiamo terminato di vedere quali sono le funzioni più utili del PHP, senza addentrarci in argomenti come database e simili. Ovviamente, la lista delle funzioni non termina qui ma, essendocene altrettante meno utili almeno per chi inizia a programmare con questo linguaggio, abbiamo preferito fermarci a questo punto. Altre specifiche le tratteremo nelle lezioni successive.

PHP offre agli sviluppatori una serie di funzioni relative alla crittazione, legate alla libreria mcrypt; tale libreria supporta moltissimi algoritmi, alcuni più utilizzati ed altri meno. Gli algoritmi sono:

- DES
- TripleDES
- Blowfish
- 3-WAY
- SAFER-SK64
- SAFER-Sk128
- TWOFISH
- TEA
- RC2
- GOST
- RC6
- IDEA

Per funzionare con tale libreria, il PHP deve essere stato compilato con l'opzione --with-mcrypt. I comandi fondamentali sono quattro, tutti con la medesima sintassi:

- `mcrypt_cfb()`: cipher feedback; codifica byte per byte;
- `mcrypt_cbc()`: cipher block chaining: utile per l'encoding dei file con un grande margine di sicurezza;
- `mcrypt_ecb()`: electronic codebook: utilizzata per dati random, dove il livello di sicurezza non è altissimo;
- `mcrypt_ofb()`: output feedback: simile a cfb, ma è data maggiore attenzione agli errori.

La sintassi in generale è:

```
$encrypted = mcrypt_XXX(algoritmo, chiave, input, encode/decode)
```

dove:

XXX è il metodo che si intende utilizzare (cfb, cbc, cfb o ofb);

algoritmo è l'algoritmo che si intende utilizzare, con la sintassi:

```
MCRYPT_ALGORITMO
```

Ad esempio, si potrebbe utilizzare

```
MCRYPT_BLOWFISH
```

oppure

```
MCRYPT_IDEA
```

chiave altro non è che la chiave con cui si andranno a crittare i dati;

input sono i dati da crittare;

encode/decode indica alla funzione se si devono crittare o decrittare i dati; per questo, si usano rispettivamente:

```
MCRYPT_ENCRYPT
```

e

```
MCRYPT_DECRYPT
```

Vediamo ora un esempio: volendo crittare una semplice stringa di testo con chiave di crittatura "La mia chiave" utilizzando CFB con l'algoritmo IDEA, dovremo scrivere:

```
$stringa = "Una semplice stringa di testo";  
$chiave = "La mia chiave";  
$encrypted = mcrypt_cfb(MCRYPT_IDEA, $chiave, $stringa, MCRIPT_ENCRYPT);
```

Chiunque voglia poi leggere i nostri dati crittati (\$encrypted) dovrà ovviamente conoscere la chiave, il metodo e l'algoritmo utilizzati; quindi potrebbe scrivere qualcosa del tipo:

```
$chiave = "La mia chiave";  
$stringa = mcrypt_cfb(MCRYPT_IDEA, $chiave, $encrypted, MCRIPT_DECRYPT);
```

La sua variabile "\$stringa", quindi, conterrà "Una semplice stringa di testo".

FTP

Fra i vari protocolli, PHP ci mette a disposizione una vasta libreria di funzioni legate al protocollo FTP (FILE TRANSFER PROTOCOL), per il trasferimento di file da un computer all'altro in una rete. Vediamone le principali.

ftp_connect

Questa è la funzione "principale" nel senso che ci permette di stabilire una connessione FTP fra la nostra macchina ed il server FTP remoto. La sua sintassi è `$stream = ftp_connect(host, port)`;

dove `host` è il nome del server a cui intendiamo connetterci e `port` (opzionale) è la porta alternativa alla quale ci si vuole connettere; se questa non è specificata, viene utilizzata la porta di default per il protocollo FTP, ossia la 21. Nella variabile `$stream`, inoltre, viene immagazzinato appunto lo stream di dati che il client (in questo caso il PHP) riceve dal server, ossia i messaggi di connessione

accettata (con i vari dettagli) o di connessione rifiutata.

Ad esempio, per connetterci alla porta di default del server FTP "ftp://ftp.host.com" utilizzeremo:

```
$stream = ftp_connect("ftp://ftp.host.com", 8023);
```

ftp_login

Dopo la connessione, abbiamo bisogno di identificarci in modo che il server ci permetta lo scambio dei dati. molti saranno abituati a non vedere tale fase visto che, con i più diffusi client FTP grafici essa è svolta in automatico utilizzando le informazioni di login (username e password) inseriti come opzioni per il collegamento, ma sappiate che questa è una fase di vitale importanza per la connessione. La sintassi della funzione è:

```
$login = ftp_login($stream, username, password);
```

Se ad esempio in precedenza ci eravamo collegati all'host "ftp.host.com", utilizzando la variabile "\$stream", adesso potremo procedere al login vero e proprio con:

```
$login = ftp_login($stream, "utente", "password");
```

La variabile \$login ci servirà per capire se il login è andato o meno a buon fine e conterrà il valore "1" per il successo, "0" altrimenti. Ad esempio, per vedere se continuare lo scambio di dati in seguito all'autorizzazione potremo utilizzare il valore assegnato a tale variabile e scrivere:

```
if ($login == "1") {  
    # Fai il resto delle operazioni  
} else {  
    echo "Autorizzazione non riuscita\n";  
}
```

Una volta connessi, potremo sapere su che macchina stiamo lavorando con la funzione ftp_systype() che ha sintassi:

```
$system = ftp_systype($stream);
```

ftp_pwd

Questa funzione invoca il comando "pwd", ovvero "Print work directory", che potremo tradurre come "Visualizza la directory corrente". Per vedere a che directory veniamo connessi dopo il login, potremo scrivere:

```
$directory = ftp_pwd($stream);
```

dove \$stream è sempre la variabile che abbiamo utilizzato per la connessione con ftp_connect().

ftp_cdup e ftp_chdir

Queste due funzioni servono rispettivamente a muoversi alla directory superiore e a muoversi in una determinata directory all'interno del server.

La prima si utilizza con sintassi:

```
$var = ftp_cdup($stream);
```

La seconda invece:

```
$newdir = ftp_chdir($stream, "nuova_directory");
```

Se ad esempio al login siamo nella directory "/" e volessimo spostarci in "/var/wwwdata" potremo scrivere:

```
$newdir = ftp_chdir($stream, "/var/wwwdata");
```

ftp_mkdir e ftp_rmdir

Queste due funzioni invocano il comando "mkdir" (crea una directory) e "rmdir" (rimuovi una directory). La prima restituisce il nome della nuova directory, la seconda solamente i valori true o false. Potremo creare un piccolo loop e scrivere:

```
# Posizioniamoci in "/var/wwwdata".
$mydir = ftp_chdir($stream, "/var/wwwdata/");

# Creiamo la directory "prova" come sottodirectory di "/var/wwwdata"
$newdir = ftp_mkdir($stream, "prova")

# Cancelliamo la directory appena creata!
$deleted_dir = ftp_rmdir($stream, $newdir);

# Possiamo ora controllare il tutto con:
if ($deleted_dir == "1") {
    print "Operazione completata con successo.\n";
} else {
    print "Qualcosa non è andato per il verso giusto.\n";
}
```

Ovviamente l'esempio non ha molto senso in una vera connessione (perchè creare una directory e subito cancellarla?), ma è stato proposto per comprendere come utilizzare al meglio queste due funzioni.

ftp_nlist

Questa funzione è analoga al comando "ls" o "dir", ossia il comando utilizzato per vedere i nomi dei file presenti in una directory. La sua sintassi è:

```
$list = ftp_nlist($stream, directory);
```

Ad esempio, possiamo portarci nella directory "/var/wwwdata" e leggerne i file con:

```
$newdir = "/var/wwwdata";
$list = ftp_nlist($stream, $newdir);
```

I risultati sono contenuti in un array, quindi un 'echo "\$list"' non avrebbe alcun senso.

ftp_get

Funzione che richiama il comando GET, per scaricare un file dal server remoto. Dobbiamo specificare per la funzione, oltre al solito stream, il nome del file locale, il nome del file remoto e la modalità di trasferimento (FTP_ASCII o FTP_BINARY); la sintassi completa è:

```
$file = ftp_get($stream, local_filename, remote_filename, mode);
```

Ad esempio, volendo scaricare dal server il file "data.txt" (supponiamo di essere già all'interno della directory che lo contiene) inserendolo nella directory "/tmp" con nome "file.txt" in ASCII mode, scriveremo:

```
$file = ftp_get($stream, "/tmp/file.txt", "data.txt", FTP_ASCII);
```

Per vedere se l'operazione ha avuto o meno successo, possiamo operare in due modi: controllare se effettivamente il file c'è nel nostro disco oppure controllare il valore della variabile \$file: se ha valore "1" allora l'operazione è stata completata, se ha valore "0" nessun file sarà stato scaricato sul nostro disco.

ftp_put

Questa funzione fa esattamente il contrario di ftp_get(), ossia carica un file sul server. La sua

sintassi è:

```
$file = ftp_put($stream, remote_filename, local_filename, mode);
```

Le opzioni sono identiche alle precedenti, quindi possiamo fare l'esempio contrario del precedente: carichiamo il file locale "/tmp/file.txt" nella directory remota (siamo già in questa directory) con il nome "data.txt". Tutto in ASCII mode, ovviamente:

```
$file = ftp_put($stream, "data.txt", "/tmp/file.txt", FTP_ASCII);
```

Anche qui, possiamo controllare in due modi: valutando il valore di \$file oppure invocando la funzione ftp_nlist() per vedere se fra i file c'è anche "data.txt".

ftp_size

Restituisce le dimensioni di un dato file. La sintassi è:

```
$size = ftp_size($stream, remote_filename);
```

Per tornare agli esempi precedentemente fatti, vediamo di conoscere la grandezza del file "data.txt", che si trova nella directory in cui siamo al momento; basterà scrivere:

```
$size = ftp_size($stream, "data.txt");
```

in modo che la variabile \$size contenga le dimensioni del file "data.txt".

ftp_mdtm

Restituisce la data di ultima modifica di un file, restituendola come Unix timestamp. La sintassi è:

```
$date = ftp_mdtm($stream, remote_filename);
```

Ad esempio, volendo sapere la data di ultima modifica del file "data.txt" possiamo scrivere:

```
$date = ftp_mdtm($stream, "data.txt");
```

Anche in questo caso, la variabile "\$date" conterrà la data di ultima modifica del file oppure il valore "-1" in caso di insuccesso (file inesistente o casi del genere).

ftp_rename e ftp_delete

Come apparirà chiaro dai nomi, queste due funzioni servono per rinominare un file e per cancellarlo. La prima ha sintassi:

```
$name = ftp_rename($stream, oldname, newname);
```

dove "oldname" è il nome originario del file e "newname" è il nuovo nome che vogliamo assegnare al file.

Ad esempio, per rinominare il file "data.txt" in "dati.dat" possiamo scrivere:

```
$name = ftp_rename($stream, "data.txt", "dati.dat");
```

La variabile \$name conterrà "1" se l'operazione ha avuto successo, "0" altrimenti (file inesistente o casi simili).

La funzione ftp_delete(), invece, si utilizza con sintassi:

```
$delete = ftp_delete($stream, file);
```

Ad esempio, per eliminare il file "dati.dat" presente nella "current-directory" possiamo scrivere:

```
$delete = ftp_delete($stream, "dati.dat");
```

Anche in questo caso la variabile può contenere valore "1" (il file è stato eliminato) o "0" (qualcosa non è andato per il verso giusto).

ftp_quit

A questo punto, il nostro lavoro sul server è terminato e possiamo disconnetterci utilizzando la funzione ftp_quit() che ha la semplice sintassi:

```
$quit = ftp_quit($stream).
```

È sempre consigliato invocare questa funzione invece di chiudere il programma in esecuzione, più che altro per una questione di rispetto verso il server.

Le estensioni

Nei precedenti capitoli abbiamo visto le principali funzioni legate al linguaggio per la creazione delle nostre pagine dinamiche. La differenza fra quelle funzioni e la famiglia di quelle che andremo a vedere è sostanziale: le prime sono presenti direttamente all'interno del motore (built-in), le seconde sono presenti in librerie aggiuntive che devono essere installate sul sistema e richiamate in maniera particolare.

Prima di tutto, vediamo di capire il meccanismo di caricamento dinamico di queste librerie: aprendo il file `php.ini` vedremo un paragrafo dedicato alle "Extension", ossia estensioni nel linguaggio stesso: per spiegarci meglio, potremo dire che queste sono un insieme di librerie che vengono richiamate al momento dell'esecuzione di uno script come avviene, in maniera analoga, per il caricamento dei moduli con un webserver.

La sintassi per il caricamento di queste estensioni di linguaggio è molto semplice: una volta che avremo installato la libreria sul sistema, non ci resta che aggiungere nel file `php.ini` la riga:

```
extension=libreria
```

È qui necessario un discorso mirato per i sistemi Unix ed i sistemi Windows: per entrambi la sintassi è identica, ma ovviamente il nome delle librerie e la loro estensione no.

Nei sistemi Windows, una libreria si riconosce dall'estensione ".dll", mentre per Unix questa è ".so": quindi, a seconda del sistema, dovremo utilizzare il corretto nome per la libreria e, soprattutto, la corretta estensione. Ovviamente, per chi abbia entrambi i sistemi installati e riesca da uno dei sistemi a vedere l'altro è chiaro che le dll non possono essere caricate su un sistema Unix e viceversa.

Nello scrivere il nome della libreria che ci interessa caricare, non dobbiamo soffermarci sul percorso completo, ma è necessario solamente il nome della stessa, ad esempio `pgsql.so` per i database Postgres. Questo perché, nello stesso file, è presente un'altra linea di configurazione che definisce in quale directory sono presenti queste librerie: leggendo il file `php.ini` potrete trovare la riga `extension_dir = directory` che instruirà il motore sulla locazione standard delle librerie. Quindi, quando specifichiamo con `extension` una libreria che vogliamo sia caricata per l'esecuzione di uno script, vengono di fatto uniti `extension_dir` ed `extension`: se ad esempio `extension_dir` è `/usr/lib/php` e abbiamo impostato `extension=pgsql.so`, il PHP saprà che per caricare la libreria `pgsql.so` dovrà cercarla in `/usr/lib/php/pgsql.so`.

Inoltre, con l'istruzione `extension=` è possibile specificare non solo una libreria ma tutta la serie di librerie che ci possono fare comodo: ad esempio possiamo avere qualcosa del tipo:

```
extension=pgsql.so
```

```
extension=mysql.so
```

```
extension=gd.so
```

```
extension=imap.so
```

```
extension=ldap.do
```

```
extension=xml.so
```

e via dicendo; come noterete dalla seconda riga, poi, è possibile specificare anche due o più librerie per uno stesso "campo" in questo caso, ci sono due librerie per due database (Postgres e MySQL) che, oltre a non entrare in conflitto l'una con l'altra, potrebbero teoricamente anche essere utilizzate contemporaneamente, a patto che questo abbia un'utilità.

Nel caso si cerchi di richiamare una funzione non built-in all'interno di uno script, lo script visualizza un messaggio d'errore che ci avverte che stiamo cercando di utilizzare una funzione non riconosciuta: ad esempio, per i database, devono essere caricate le estensioni come detto prima e, solo dopo aver compiuto questo passo, sarà possibile utilizzare le funzioni ad essi relativi senza incorrere in messaggi d'errore, sempre che queste siano utilizzate in maniera corretta, ovviamente.

Il perchè delle estensioni

A questo punto, ci sarà sicuramente chi si chiederà il perchè di questa librerie aggiuntive (a volte molto importanti o addirittura essenziali) che devono essere scaricate, installate e richiamate indirettamente. La spiegazione è semplice: per non appesantire inutilmente il motore. Di per sè questo ha già un grande numero di funzioni built-in, che potremo definire le funzioni "principali" per il linguaggio, sebbene alcune possano sembrare di poca utilità. Immaginate ora che il motore avesse al suo interno anche tutte le funzioni relative a tutti i database che supporta: avrebbe un altro centinaio (abbondante) di funzioni al suo interno; e questo non sarebbe un male se il 95% di queste non fosse inutilizzato.

Chi, infatti, ha la necessità di lavorare con il PHP ed una decina di database differenti? È sicuramente meglio installare l'estensione per il database che si deve utilizzare e non appesantire il motore con funzioni che certamente non utilizzeremo mai. Inoltre, pensate anche alle funzioni della libreria GD: senza dubbio sono interessanti per i risultati che permettono di ottenere, ma quanti in realtà le utilizzano? È più semplice fornire tutte queste funzioni in un file separato e lasciare che questo venga installato ed utilizzato da chi ne ha veramente bisogno.

Se ci pensate, è quello che avviene per tutti i linguaggi di programmazione: esistono le primitive (che, in linea di massima, possiamo definire come le funzioni essenziali e built-in in un sistema) e le funzioni in qualche modo esterne che vengono richiamate all'occorrenza. Rimanendo nei linguaggi di scripting, un paragone può essere fatto ad esempio con il Perl: anch'esso ha una nutrita schiera di funzioni built-in nell'interprete, alle quali si aggiungono la quasi infinità di moduli che il programmatore può utilizzare all'interno delle proprie creazioni.

La programmazione a oggetti.

Con l'avvento di PHP 5 il modo di concepire la programmazione ad oggetti in PHP è cambiato radicalmente. Il modello ad oggetti semplificato che era presente fino alla versione 4 non è che un'ombra scialba di quello attuale. Prima gli oggetti erano solamente una funzionalità di supporto poco utilizzata e veramente troppo poco flessibile. Ora gli sviluppatori possono puntare su un supporto alla programmazione ad oggetti che avvicina il linguaggio ai concorrenti più blasonati.

In PHP 5 la definizione di una classe rispecchia molto più da vicino le esigenze degli sviluppatori enterprise. Vediamo di analizzare la sintassi utilizzata con dei brevi esempi.

La definizione di una classe avviene utilizzando una sintassi simile a quella precedente, anche se possiamo notare grosse differenze. In primo luogo il costruttore della classe ora non deve avere lo stesso nome della classe stessa ma deve chiamarsi `__construct`; in secondo luogo abbiamo la possibilità di definire dei distruttori (implementati nel metodo `__destruct`) che verranno chiamati ogni qualvolta il garbage collector di PHP distruggerà l'oggetto. Oltre a queste piccole differenze, finalmente è possibile specificare la visibilità di metodi ed attributi attraverso le parole chiave `public`, `protected` e `private`. Vediamo un semplice esempio di classe:

```
<?php
class Prova
{
    public $attr_public;
    private $attr_private;
    protected $attr_protected;

    public function __construct()
    {
        // operazioni di inizializzazione
    }

    public function __destruct()
    {
```

```

        // operazioni eseguite prima della distruzione
    }

    public function publicMethod()
    {
    }

    protected function protectedMethod()
    {
    }

    private function privateMethod()
    {
    }
}
?>

```

Oltre queste differenze ora è possibile (e necessario) specificare esplicitamente quali metodi o proprietà dovranno essere statiche. Solo le proprietà o i metodi statici possono essere acceduti utilizzando l'operatore ::, mentre prima PHP permetteva l'utilizzo di quell'operatore per qualunque metodo.

Quindi la definizione di un metodo proprietà statica presuppone l'anteposizione della parola chiave static prima della definizione.

Ovviamente l'ereditarietà è ancora supportata, ma grazie alle aggiunte è notevolmente migliorata. Difatti possiamo sfruttare la possibilità di definire la visibilità di metodi e proprietà per facilitare il design dell'applicazione.

```

<?php

class ExtProva extends Prova
{
    public function __construct()
    {
        parent::__construct();
    }

    private function provaProtected()
    {
        $this-> protectedMethod();
    }
}

?>

```

Ogni definizione di classe inizia con la parola chiave class, seguita dal nome della classe e da una coppia di parentesi graffe che ne contengono la definizione dei membri e di metodi. Viene resa disponibile anche una pseudo variabile ogni qual volta il metodo venga chiamata da un oggetto. \$this è un riferimento all'oggetto chiamante, che può essere un oggetto diverso da quello a cui appartiene il metodo se il secondo oggetto chiama il metodo del primo in maniera statica (cioè come metodo di classe). Nell'esempio che segue viene spiegata questa affermazione:

```

<?php
class A
{

```

```

function foo()
{
    if (isset($this)) {
        echo '$this is defined (';
        echo get_class($this);
        echo ")\n";
    } else {
        echo "\$this is not defined.\n";
    }
}

class B
{
    function bar()
    {
        A::foo();
    }
}

$a = new A();
$a->foo();
A::foo();
$b = new B();
$b->bar();
B::bar();
?>

```

Il precedente esempio visualizzerà:

```

$this is defined (a)
$this is not defined.
$this is defined (b)
$this is not defined.

```

Per creare l'istanza di un oggetto, ne deve essere creato uno e assegnato ad una variabile, per fare ciò si usa la funzione new.

Overloading

Sia le chiamate di metodi che l'accesso ai membri può essere sovraccaricato (overloaded) attraverso i metodi `__call`, `__get`, `__set`, `__isset`, `__unset`. Questi vengono lanciati solo quando il tuo oggetto non contiene il membro o il metodo a cui stai cercando di accedere. Tutti i metodi overloaded non possono essere definiti come statici.

Overload dei membri

```

void __set ( string name, mixed value )
mixed __get ( string name )
bool __isset ( string name )
void __unset ( string name )

```

Membri di classe possono essere sovraccaricati per eseguire codice personalizzato definito nella tua classe.

```

<?php
class Setter

```

```

{
public $n;
private $x = array("a" => 1, "b" => 2, "c" => 3);

private function __get($nm)
{
    echo "Getting [$nm]\n";

    if (isset($this->x[$nm])) {
        $r = $this->x[$nm];
        print "Returning: $r\n";
        return $r;
    } else {
        echo "Nothing!\n";
    }
}

private function __set($nm, $val)
{
    echo "Setting [$nm] to $val\n";

    if (isset($this->x[$nm])) {
        $this->x[$nm] = $val;
        echo "OK!\n";
    } else {
        echo "Not OK!\n";
    }
}

private function __isset($nm)
{
    echo "Checking if $nm is set\n";

    return isset($this->x[$nm]);
}

private function __unset($nm)
{
    echo "Unsetting $nm\n";

    unset($this->x[$nm]);
}
}

$foo = new Setter();
$foo->n = 1;
$foo->a = 100;
$foo->a++;
$foo->z++;

var_dump(isset($foo->a)); //true
unset($foo->a);
var_dump(isset($foo->a)); //false

```

```
// this doesn't pass through the __isset() method
// because 'n' is a public property
var_dump(isset($foo->n));

var_dump($foo);
?>
```

Il precedente esempio visualizzerà:

```
Setting [a] to 100
OK!
Getting [a]
Returning: 100
Setting [a] to 101
OK!
Getting [z]
Nothing!
Setting [z] to 1
Not OK!

Checking if a is set
bool(true)
Unsetting a
Checking if a is set
bool(false)
bool(true)

object(Setter)#1 (2) {
  ["n"]=>
  int(1)
  ["x:private"]=>
  array(2) {
    ["b"]=>
    int(2)
    ["c"]=>
    int(3)
  }
}
```

Overload dei metodi

mixed __call (string name, array arguments)

I metodi di classe possono essere sovraccaricati per eseguire codice personalizzato definito nella classe definendo questo metodo speciale. Il parametro di nome usato è il nome della funzione richiesta. Gli argomenti passati nella funzione verranno definiti come un array nel parametro arguments. Il valore restituito dal metodo __call() sarà restituito al chiamante.

```
<?php
class Caller
{
  private $x = array(1, 2, 3);

  public function __call($m, $a)
  {
    print "Method $m called:\n";
  }
}
```

```

    var_dump($a);
    return $this->x;
}
}

$foo = new Caller();
$a = $foo->test(1, "2", 3.4, true);
var_dump($a);
?>

```

Il precedente esempio visualizzerà:

Method test called:

```

array(4) {
  [0]=>
  int(1)
  [1]=>
  string(1) "2"
  [2]=>
  float(3.4)
  [3]=>
  bool(true)
}
array(3) {
  [0]=>
  int(1)
  [1]=>
  int(2)
  [2]=>
  int(3)
}

```

Le classi astratte

Oltre a questo, PHP ha ampliato notevolmente le sue potenzialità introducendo le interfacce e le classi astratte. Un'interfaccia è una classe speciale che definisce solamente la struttura che dovranno obbligatoriamente avere le classi che la implementano. Non può essere istanziata.

```

<?php

interface ProvaInterfaccia
{
    public function mustBeImplemented();
}

class Prova implements ProvaInterfaccia
{
    public function mustBeImplemented()
    {
        // implementazione
    }
}

?>

```

Se una classe non dovesse implementare tutti i metodi definiti nell'interfaccia, PHP restituirebbe un errore.

Le classi astratte invece sono un sistema che permette di definire classi parzialmente completate che lasciano l'implementazione di alcuni metodi alle sottoclassi. Una classe astratta deve essere definita utilizzando la parola chiave `abstract`; lo stesso vale per quei metodi astratti della classe. Vediamo un esempio:

```
<?php
abstract class ProvaAbs
{
    public function prova()
    {
        $this->abstractMethod();
    }

    abstract protected function abstractMethod();
}

class Prova extends ProvaAbs
{
    protected function abstractMethod()
    {
        // metodo richiamato da ProvaAbs::prova()
    }
}
?>
```

Grazie alle interfacce ed alle classi astratte, PHP ha rafforzato notevolmente il suo supporto ai tipi di dato, permettendo l'introduzione del `type hinting` per i tipi di dato complessi. Possiamo specificare il tipo di oggetto che ci aspettiamo per un parametro di un metodo/funzione demandando a PHP il compito di effettuare il controllo:

```
<?php
function prova(ProvaAbs $arg)
{
    $arg->prova();
}
?>
```

In questo modo possiamo assicurarci un'integrità delle chiamate che prima era quasi impossibile se non con particolari controlli effettuati a runtime.

Costanti di classe e altre funzionalità

Un'altra aggiunta interessante è che ora è possibile definire costanti di classe, accedendovi utilizzando l'operatore usato per le proprietà o i metodi statici:

```
<?php
class Prova
{
```

```
    const MIA_COSTANTE = 'valore';
}

echo Prova::MIA_COSTANTE;

?>
```

Prima di terminare questa breve trattazione della programmazione ad oggetti, vorrei parlare di un'aggiunta fatta a PHP che permette di caricare automaticamente i file contenenti le definizioni delle classi nel momento in cui non vengano trovate in fase di esecuzione. PHP richiama automaticamente la funzione `__autoload` quando non trova una classe:

```
<?php

function __autoload($classname)
{
    echo Classe richiesta: $classname;
    require_once $classname.php;
}

$prova = new Prova();

?>
```

Ovviamente utilizzare direttamente il nome della classe non è un sistema corretto. Ricordiamoci sempre di controllare che il path risultante o richiesto sia realmente accessibile e sia corretta la sua richiesta.

Nuove classi built-in

Oltre al potenziamento del modello ad oggetti, gli sviluppatori hanno deciso di aggiungere alla quinta versione di PHP una serie di classi built-in molto interessanti. Queste classi arricchiscono il comportamento di PHP e la potenzialità del suo modello ad oggetti aggiungendo il supporto per strutture dato molto potenti derivate dai pattern di sviluppo più comuni ed utilizzati.

Il primo che ho deciso di trattare è il pattern Iterator. Un iteratore è un oggetto che si comporta come una lista da cui è possibile recuperare ciclicamente gli elementi, con la differenza che la lista non è (normalmente) già salvata in memoria ma gli elementi successivi a quello corrente vengono generati su richiesta. Questo permette iterazioni su liste virtualmente infinite o di cui non si conoscono a priori le dimensioni. Gli iteratori possono essere utilizzati in congiunzione con il costrutto `foreach` come se si trattasse di normali array.

Vediamo un esempio molto semplice di iteratore in PHP:

```
<?php

class MiaListaIterator implements Iterator
{
    private $array;
    private $valid;

    public function __construct($array)
    {
        $this->array = $array;
        $this->valid = false;
    }
}
```

```

    public function rewind()
    {
        $this->valid = (FALSE !== reset($this->array));
    }

    public function current()
    {
        return current($this->array);
    }

    public function key()
    {
        return key($this->array);
    }

    public function valid()
    {
        return $this->valid;
    }

    public function next()
    {
        $this->valid = (FALSE !== next($this->array));
    }
}

class MiaLista implements IteratorAggregate
{
    private $range;

    public function __construct($max)
    {
        $this->range = range(0, $max);
    }

    public function getIterator()
    {
        return new MiaListaIterator($this->range);
    }
}

$prova = new MiaLista(500);
foreach($prova as $item)
{
    echo $item.<br />;
}

?>

```

Come potete notare dal codice intervengono le interfacce Iterator ed IteratorAggregate. La prima serve per permettere a PHP di assicurarsi che l'iteratore recuperato segua una struttura precisa che permette di operare sull'elemento come se fosse un array (recuperando la chiave corrente, il prossimo elemento, l'elemento corrente, o riavvolgendo), mentre la seconda serve per assicurarsi che ogni oggetto utilizzato da foreach come un'array possa restituire un iteratore in modo corretto,

tramite getIterator.

L'esempio in realtà non è molto significativo perchè lo stesso comportamento potrebbe essere ottenuto utilizzando un normale array. Ma pensiamo al recupero di memoria che potrebbe avvenire in questo caso:

```
<?php
class MiaListaIterator implements Iterator
{
    private $max;
    private $valid;
    private $current;

    public function __construct($max)
    {
        $this->max = $max;
        $this->current = 0;
        $this->valid = true;
    }

    public function rewind()
    {
        $this->current = 0;
        $this->valid = true;
    }

    public function current()
    {
        return $this->current;
    }

    public function key()
    {
        return $this->current;
    }

    public function valid()
    {
        return $this->valid;
    }

    public function next()
    {
        $this->valid = ++$this->current > $this->max;
    }
}

class MiaLista implements IteratorAggregate
{
    private $range;

    public function __construct($max)
    {
        $this->range = $max;
    }
}
```

```

        public function getIterator()
        {
            return new MiaListaIterator($this->range);
        }
    }

    $prova = new MiaLista(5000000000);
    foreach($prova as $item)
    {
        echo $item.<br />;
    }
?>

```

Con questo codice iteriamo su un numero molto elevato di elementi senza consumare eccessiva memoria. Sarebbe stato molto duro ottenere lo stesso risultato utilizzando un array di cinque miliardi di elementi. La SPL implementa iteratori per molte tipologie di dato e strutture (array, directory e molto altro), quindi consiglio di studiarla per evitare di ripetere le operazioni già fatte nativamente da altri.

Un'altra classe che ritengo molto interessante è `ArrayObject`, che permette ad un oggetto di assumere il comportamento di un normale array, con la possibilità di accedere ai suoi elementi utilizzando le parentesi quadre.

```

<?php

class Users extends ArrayObject
{
    private $db;

    public function __construct($db)
    {
        $this->db = $db;
    }

    public function offsetGet($index)
    {
        $result = $this->db->query("SELECT * FROM users WHERE name =
'" . $index . "'");
        return $result->fetch_assoc();
    }

    public function offsetSet($index, $value)
    {
        $this->db->query("UPDATE users
        SET surname = '" . $value . "'
        WHERE name = '" . $index . "'");
    }

    public function offsetExists($index)
    {
        $result = $this->db->query("SELECT * FROM users WHERE name =
'" . $index . "'");
        return $result->num_rows > 0;
    }
}

```

```

    }

    public function offsetUnset($index)
    {
        $this->db->query("DELETE FROM users WHERE name = '". $index. "'");
    }
}

$utenti = new Users;
echo $utenti['Gabriele'];
$utenti['Paolo'] = 'Rossi';
unset($utenti['Federico']);

?>

```

Questo esempio vi fa comprendere il funzionamento di ArrayObject: una volta implementati i metodi sopra elencati, possiamo tranquillamente operare sull'istanza dell'oggetto come se fosse un'array. Ovviamente sconsiglio di utilizzare un sistema simile per gestire gli utenti, ma ArrayObject può risultare molto utile in molte situazioni di programmazione avanzata.

Eseguire programmi esterni in php

Operatori di esecuzione

PHP supporta un operatore di esecuzione: backticks (`). Notare che quelli non sono apostrofi! PHP cercherà di eseguire il contenuto dei backticks come comando di shell; sarà restituito l'output (i.e., non sarà semplicemente esportato come output; può essere assegnato ad una variabile). L'uso dell'operatore backtick è identico alla funzione `shell_exec()`.

```

<?php
$output = `ls -al`;
echo "<pre>$output</pre>";
?>

```

Nota: L'operatore backtick è disabilitato quando è abilitata modalità sicura oppure quando è disabilitata `shell_exec()`.

In modalità sicura i programmi esterni richiamabili dagli script php devono essere per forza nella directory definita nel file di inizializzazione di php, alla voce `safe_mode_exec_dir`, deve essere usato il / come separatore delle directory anche in windows.

exec

`string exec (string command [, array &output [, int &return_var]])`

`exec()` esegue il comando passato da `command`, la funzione non invia nessun output. Restituisce semplicemente l'ultima linea dal risultato del comando. Se si ha bisogno di eseguire un comando ed avere tutti i dati passati direttamente indietro senza alcuna interferenza, usare la funzione `passthru()`. Se l'argomento `output` è presente, allora tale vettore specificato verrà riempito con ogni linea del output del comando. I fine riga, come `\n` non sono inclusi in questo array. Notare che se il vettore contiene già degli elementi, `exec()` li aggiungerà in coda al vettore. Se non si vuole che la funzione aggiunga elementi, eseguire un `unset()` sul vettore prima di passarlo ad `exec()`.

Se viene passato l'argomento `return_var` assieme all'argomento `output`, allora lo stato del comando eseguito verrà scritto in questa variabile.

```

<?php
// restituisce la username del proprietario del processo php/httpd attivo

```

```
// (su un sistema con l'eseguibile "whoami" nel path)
echo exec('whoami');
?>
```

Avvertimento

Se si permette di passare a questa funzione i dati provenienti dagli input utente, si dovrebbe utilizzare la funzione `escapeshellarg()` oppure `escapeshellcmd()` in modo da essere sicuri che gli utenti non possano compromettere il sistema eseguendo comandi arbitrari.

Nota: Se si vuole avviare un programma tramite questa funzione e lasciarlo girare in background, occorre essere certi che l'output del programma sia rediretto su un file o qualche altro flusso di output altrimenti il PHP si sospenderà fino a quando il programma non termina.

Nota: Quando si abilita la modalità sicura, si può eseguire soltanto gli eseguibili presenti nella directory `safe_mode_exec_dir`. Per motivi pratici, attualmente, non è permesso avere .. come componente del percorso di un eseguibile.

Avvertimento

Con la modalità sicura attivata, tutte le parole che seguono il comando iniziale sono trattate come argomenti. Quindi, `echo y | echo x` diventa `echo "y | echo x"`.

passthru

```
void passthru ( string command [, int &return_var] )
```

La funzione `passthru()` è simile alla funzione `exec()` in quanto esegue `command`. Se il parametro `return_var` è specificato, lo stato ritornato dal comando Unix verrà posto lì. Questa funzione deve essere usata al posto di `exec()` o di `system()` quando l'output del comando Unix consiste in dati binari da passare direttamente al browser. Un suo uso frequente consiste nel eseguire, ad esempio, le utility `pbmplus` che possono restituire un flusso diretto all'immagine. Impostando il tipo di contenuto a `image/gif` e successivamente chiamando un programma `pbmplus` per generare una gif puoi realizzare uno script PHP che genera direttamente immagini.

Avvertimento

Se si permette di passare a questa funzione i dati proveninetti dagli input utente, si dovrebbe utilizzare la funzione `escapeshellarg()` oppure `escapeshellcmd()` in modo da essere sicuri che gli utenti non possano compromettere il sistema eseguendo comandi arbitrari.

Nota: Se si vuole avviare un programma tramite questa funzione e lasciarlo girare in background, occorre essere certi che l'output del programma sia rediretto su un file o qualche altro flusso di output altrimenti il PHP sarà sospenderà fino a quando il programma non termina.

Nota: Quando si abilita la modalità sicura, si può eseguire soltanto gli eseguibili presenti nella directory `safe_mode_exec_dir`. Per motivi pratici, attualmente, non è permesso avere .. come componente del percorso di un eseguibile.

Avvertimento

Con la modalità sicura attivata, tutte le parole che seguono il comando iniziale sono trattate come argomenti. Quindi, `echo y | echo x` diventa `echo "y | echo x"`.

system

```
string system ( string command [, int &return_var] )
```

`system()` è semplicemente come la versione C della funzione che esegue il `command` dato e restituisce in uscita il risultato. Se viene fornita una variabile come secondo argomento, allora il codice di stato ritornato dal comando eseguito verrà scritto in tale variabile.

Avvertimento

Se si permette di passare a questa funzione i dati proveninetti dagli input utente, si dovrebbe

utilizzare la funzione `escapeshellarg()` oppure `escapeshellcmd()` in modo da essere sicuri che gli utenti non possano compromettere il sistema eseguendo comandi arbitrari.

Nota: Se si vuole avviare un programma tramite questa funzione e lasciarlo girare in background, occorre essere certi che l'output del programma sia rediretto su un file o qualche altro flusso di output altrimenti il PHP sarà sospenderà fino a quando il programma non termina.

La chiamata a `system()` tenta anche di ripulire automaticamente il buffer di output del web server dopo ogni linea di output se PHP gira come un modulo server.

Restituisce l'ultima linea del output del comando se ha successo e FALSE se fallisce.

Se devi eseguire un comando ottenendo tutti i dati restituiti dal comando direttamente senza alcuna interferenza, usa la funzione `passthru()`.

```
<?php
echo '<pre>';

// Mette in output tutti i risultati della shellcommand "ls", e restituisce
// l'ultima linea di output nella $last_line. Memorizza il valore restituito
// del comando da shell in $retval.
$last_line = system('ls', $retval);

// Stampa informazioni aggiuntive
echo '
</pre>
<hr />L'ultima linea dell'output: ' . $last_line . '
<hr />Restituisce il valore: ' . $retval;
?>
```

Nota: Quando si abilita la modalità sicura, si può eseguire soltanto gli eseguibili presenti nella `directory safe_mode_exec_dir`. Per motivi pratici, attualmente, non è permesso avere `..` come componente del percorso di un eseguibile.

Avvertimento

Con la modalità sicura attivata, tutte le parole che seguono il comando iniziale sono trattate come argomenti. Quindi, `echo y | echo x` diventa `echo "y | echo x"`.

shell_exec

Descrizione

`string shell_exec (string cmd)`

Questa funzione è identica all'operatore backtick.

```
<?php
$output = shell_exec('ls -lart');
echo "<pre>$output</pre>";
?>
```

Nota: Questa funzione è disabilitata nella modalità `safe-mode`

Esempio di programma per lanciare un comando esterno

```
<?php
$runCommand = 'php -q FULLPATH/FILE.php';

if(isset($_SERVER['PWD']))/*nix (aka NOT windows)
{
    $nullResult = `$runCommand > /dev/null &`;
}
}
```

```
else //windows
{
    $WshShell = new COM("WScript.Shell");
    $oExec = $WshShell->Run($runCommand, 7, false);
}
?>
```