PHP Manual

Stig Sæther Bakken

Alexander Aulbach

Egon Schmid

Jim Winstead

Lars Torben Wilson

Rasmus Lerdorf

Andrei Zmievski

Jouni Ahto

Edited by

Stig Sæther Bakken

Egon Schmid

29-09-2002

PHP Manual

by Stig Sæther Bakken, Alexander Aulbach, Egon Schmid, Jim Winstead, Lars Torben Wilson, Rasmus Lerdorf, Andrei Zmievski, and Jouni Ahto

Edited by Stig Sæther Bakken and Egon Schmid

Published 29-09-2002

Table of Contents

Preface

PHP, which stands for "PHP: Hypertext Preprocessor" is a widely-used Open Source general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. Its syntax draws upon C, Java, and Perl, and is easy to learn. The main goal of the language is to allow web developers to write dynamically generated webpages quickly, but you can do much more with PHP.

This manual consists primarily of a function reference, but also contains a language reference, explanations of some of PHP’s major features, and other supplemental information.

You can download this manual in several formats at http://www.php.net/docs.php. The downloads are updated as the content changes. More information about how this manual is developed can be found in the ’About the manual’ appendix.

Part I. Getting Started

Chapter 1. Introduction

What is PHP?

PHP (recursive acronym for "PHP: Hypertext Preprocessor") is a widely-used Open Source general-purpose scripting language that is especially suited for Web development and can be embedded into HTML.

Simple answer, but what does that mean? An example:

Example 1-1. An introductory example

```
<html>
    <head>
        <title>Example</title>
    </head>
    <body>

        <?php
        echo "Hi, I’m a PHP script!";
        ?>

    </body>
</html>
```

Notice how this is different from a script written in other languages like Perl or C -- instead of writing a program with lots of commands to output HTML, you write an HTML script with some embedded code to do something (in this case, output some text). The PHP code is enclosed in special start and end tags that allow you to jump into and out of "PHP mode".

What distinguishes PHP from something like client-side JavaScript is that the code is executed on the server. If you were to have a script similar to the above on your server, the client would receive the results of running that script, with no way of determining what the underlying code may be. You can even configure your web server to process all your HTML files with PHP, and then there’s really no way that users can tell what you have up your sleeve.

The best things in using PHP are that it is extremely simple for a newcomer, but offers many advanced features for a professional programmer. Don’t be afraid reading the long list of PHP’s features. You can jump in, in a short time, and start writing simple scripts in a few hours.

Although PHP’s development is focused on server-side scripting, you can do much more with it. Read on, and see more in the What can PHP do? section.

What can PHP do?

Anything. PHP is mainly focused on server-side scripting, so you can do anything any other CGI program can do, such as collect form data, generate dynamic page content, or send and receive cookies. But PHP can do much more.

There are three main fields where PHP scripts are used.

- Server-side scripting. This is the most traditional and main target field for PHP. You need three things to make this work. The PHP parser (CGI or server module), a webserver and a web browser. You need to run the webserver, with a connected PHP installation. You can access the PHP program output with a web browser, viewing the PHP page through the server. See the installation instructions section for more information.
- Command line scripting. You can make a PHP script to run it without any server or browser. You only need the PHP parser to use it this way. This type of usage is ideal for scripts regularly executed using cron (on *nix or Linux) or Task Scheduler (on Windows). These scripts can also be used for simple text processing tasks. See the section about Command line usage of PHP for more information.
- Writing client-side GUI applications. PHP is probably not the very best language to write windowing applications, but if you know PHP very well, and would like to use some advanced PHP features in your client-side applications you can also use PHP-GTK to write such programs. You also have the ability to write cross-platform applications this way. PHP-GTK is an extension to PHP, not available in the main distribution. If you are interested in PHP-GTK, visit it's own website (http://gtk.php.net/).

PHP can be used on all major operating systems, including Linux, many Unix variants (including HP-UX, Solaris and OpenBSD), Microsoft Windows, Mac OS X, RISC OS, and probably others. PHP has also support for most of the web servers today. This includes Apache, Microsoft Internet Information Server, Personal Web Server, Netscape and iPlanet servers, Oreilly Website Pro server, Caudium, Xitami, OmniHTTPd, and many others. For the majority of the servers PHP has a module, for the others supporting the CGI standard, PHP can work as a CGI processor.

So with PHP, you have the freedom of choosing an operating system and a web server. Furthermore, you also have the choice of using procedural programming or object oriented programming, or a mixture of them. Although not every standard OOP feature is realized in the current version of PHP, many code libraries and large applications (including the PEAR library) are written only using OOP code.

With PHP you are not limited to output HTML. PHP's abilities includes outputting images, PDF files and even Flash movies (using libswf and Ming) generated on the fly. You can also output easily any text, such as XHTML and any other XML file. PHP can autogenerate these files, and save them in the file system, instead of printing it out, forming a server-side cache for your dynamic content.

One of the strongest and most significant feature in PHP is its support for a wide range of databases. Writing a database-enabled web page is incredibly simple. The following databases are currently supported:

Adabas D	Ingres	Oracle (OCI7 and OCI8)
dBase	InterBase	Ovrimos
Empress	FrontBase	PostgreSQL
FilePro (read-only)	mSQL	Solid
Hyperwave	Direct MS-SQL	Sybase
IBM DB2	MySQL	Velocis
Informix	ODBC	Unix dbm

We also have a DBX database abstraction extension allowing you to transparently use any database supported by that extension. Additionally PHP supports ODBC, the Open Database Connection standard, so you can connect to any other database supporting this world standard.

PHP also has support for talking to other services using protocols such as LDAP, IMAP, SNMP, NNTP, POP3, HTTP, COM (on Windows) and countless others. You can also open raw network sockets and interact using any other protocol. PHP has support for the WDDX complex data exchange between virtually all Web programming languages. Talking about interconnection, PHP has support for instantiation of Java objects and using them transparently as PHP objects. You can also use our CORBA extension to access remote objects.

PHP has extremely useful text processing features, from the POSIX Extended or Perl regular expressions to parsing XML documents. For parsing and accessing XML documents, we support the SAX and DOM standards. You can use our XSLT extension to transform XML documents.

While using PHP in the ecommerce field, you'll find the Cybercash payment, CyberMUT, VeriSign Payflow Pro and CCVS functions useful for your online payment programs.

At last but not least, we have many other interesting extensions, the mnoGoSearch search engine functions, the IRC Gateway functions, many compression utilities (gzip, bz2), calendar conversion, translation...

As you can see this page is not enough to list all the features and benefits PHP can offer. Read on in the sections about installing PHP, and see the function reference part for explanation of the extensions mentioned here.

Chapter 2. A simple tutorial

Here we would like to show the very basics of PHP in a short simple tutorial. This text only deals with dynamic webpage creation with PHP, though PHP is not only capable of creating webpages. See the section titled What can PHP do for more information.

PHP-enabled web pages are treated just like regular HTML pages and you can create and edit them the same way you normally create regular HTML pages.

What do I need?

In this tutorial we assume that your server has support for PHP activated and that all files ending in `.php` are handled by PHP. On most servers this is the default extension for PHP files, but ask your server administrator to be sure. If your server supports PHP then you don't need to do anything. Just create your `.php` files and put them in your web directory and the server will magically parse them for you. There is no need to compile anything nor do you need to install any extra tools. Think of these PHP-enabled files as simple HTML files with a whole new family of magical tags that let you do all sorts of things.

Your first PHP-enabled page

Create a file named `hello.php` under your webserver root directory with the following content:

Example 2-1. Our first PHP script: `hello.php`

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 <?php echo "Hello World<p>"; ?>
 </body>
</html>
```

The output of this script will be:

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 Hello World<p>
 </body>
</html>
```

Note that this is not like a CGI script. The file does not need to be executable or special in any way. Think of it as a normal HTML file which happens to have a set of special tags available to you that do a lot of interesting things.

This program is extremely simple and you really didn't need to use PHP to create a page like this. All it does is display: `Hello World` using the PHP echo() statement.

If you tried this example and it didn't output anything, or it prompted for download, or you see the whole file as text, chances are that the server you are on does not have PHP enabled. Ask your administrator to enable it for you using the Installation chapter of the manual. If you want to develop PHP scripts locally, see the downloads (http://www.php.net/downloads.php) section. You can develop locally on any Operating system, be sure to install an appropriate web server too.

The point of the example is to show the special PHP tag format. In this example we used `<?php` to indicate the start of a PHP tag. Then we put the PHP statement and left PHP mode by adding the closing tag, `?>`. You may jump in and out of PHP mode in an HTML file like this all you want.

A Note on Text Editors: There are many text editors and Integrated Development Environments (IDEs) that you can use to create, edit and manage PHP files. A partial list of these tools is maintained at PHP Editor's List (http://www.itworks.demon.co.uk/phpeditors.htm). If you wish to recommend an editor, please visit the above page and ask the page maintainer to add the editor to the list.

A Note on Word Processors: Word processors such as StarOffice Writer, Microsoft Word and Abiword are not good choices for editing PHP files.

If you wish to use one for this test script, you must ensure that you save the file as PLAIN TEXT or PHP will not be able to read and execute the script.

A Note on Windows Notepad: If you are writing your PHP scripts using Windows Notepad, you will need to ensure that your files are saved with the .php extension. (Notepad adds a .txt extension to files automatically unless you take one of the following steps to prevent it.)

When you save the file and are prompted to provide a name for the file, place the filename in quotes (i.e. "hello.php").

Alternately, you can click on the 'Text Documents' drop-down menu in the save dialog box and change the setting to "All Files". You can then enter your filename without quotes.

Something Useful

Let's do something a bit more useful now. We are going to check what sort of browser the person viewing the page is using. In order to do that we check the user agent string that the browser sends as part of its HTTP request. This information is stored in a variable. Variables always start with a dollar-sign in PHP. The variable we are interested in right now is `$_SERVER["HTTP_USER_AGENT"]`.

> **PHP Autoglobals Note:** $_SERVER is a special reserved PHP variable that contains all web server information. It's known as an Autoglobal (or Superglobal). See the related manual page on Autoglobals for more information. These special variables were introduced in PHP 4.1.0 (http://www.php.net/release_4_1_0.php). Before this time, we used the older `$HTTP_*_VARS` arrays instead, such as `$HTTP_SERVER_VARS`. Although deprecated, these older variables still exist. (See also the note on old code.)

To display this variable, we can simply do:

Example 2-2. Printing a variable (Array element)

```
<?php echo $_SERVER["HTTP_USER_AGENT"]; ?>
```

A sample output of this script may be:

```
Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
```

There are many types of variables available in PHP. In the above example we printed an Array element. Arrays can be very useful.

`$_SERVER` is just one variable that's automatically made available to you by PHP. A list can be seen in the Reserved Variables section of the manual or you can get a complete list of them by creating a file that looks like this:

Example 2-3. Show all predefined variables with phpinfo()

```
<?php phpinfo(); ?>
```

If you load up this file in your browser you will see a page full of information about PHP along with a list of all the variables available to you.

You can put multiple PHP statements inside a PHP tag and create little blocks of code that do more than just a single echo. For example, if we wanted to check for Internet Explorer we could do something like this:

Example 2-4. Example using control structures and functions

```
<?php
if (strstr($_SERVER["HTTP_USER_AGENT"], "MSIE")) {
```

```
echo "You are using Internet Explorer<br />";
}
?>
```

A sample output of this script may be:

```
You are using Internet Explorer<br />
```

Here we introduce a couple of new concepts. We have an if statement. If you are familiar with the basic syntax used by the C language this should look logical to you. If you don't know enough C or some other language where the syntax used above is used, you should probably pick up any introductory PHP book and read the first couple of chapters, or read the Language Reference part of the manual. You can find a list of PHP books at http://www.php.net/books.php.

The second concept we introduced was the strstr() function call. strstr() is a function built into PHP which searches a string for another string. In this case we are looking for `"MSIE"` inside `$_SERVER["HTTP_USER_AGENT"]`. If the string is found, the function returns `TRUE` and if it isn't, it returns `FALSE`. If it returns `TRUE`, the if statement evaluates to `TRUE` and the code within its {braces} is executed. Otherwise, it's not. Feel free to create similar examples, with if, else, and other functions such as strtoupper() and strlen(). Each related manual page contains examples too.

We can take this a step further and show how you can jump in and out of PHP mode even in the middle of a PHP block:

Example 2-5. Mixing both HTML and PHP modes

```
<?php
if (strstr($_SERVER["HTTP_USER_AGENT"], "MSIE")) {
?>
<h3>strstr must have returned true</h3>
<center><b>You are using Internet Explorer</b></center>
<?php
} else {
?>
<h3>strstr must have returned false</h3>
<center><b>You are not using Internet Explorer</b></center>
<?php
}
?>
```

A sample output of this script may be:

```
<h3>strstr must have returned true</h3>
```

```
<center><b>You are using Internet Explorer</b></center>
```

Instead of using a PHP echo statement to output something, we jumped out of PHP mode and just sent straight HTML. The important and powerful point to note here is that the logical flow of the script remains intact. Only one of the HTML blocks will end up getting sent to the viewer depending on if strstr() returned `TRUE` or `FALSE` In other words, if the string `MSIE` was found or not.

Dealing with Forms

One of the most powerful features of PHP is the way it handles HTML forms. The basic concept that is important to understand is that any form element in a form will automatically be available to your PHP scripts. Please read the manual section on Variables from outside of PHP for more information and examples on using forms with PHP. Here's an example HTML form:

Example 2-6. A simple HTML form

```
<form action="action.php" method="POST">
 Your name: <input type="text" name="name" />
 Your age: <input type="text" name="age" />
 <input type="submit">
</form>
```

There is nothing special about this form. It is a straight HTML form with no special tags of any kind. When the user fills in this form and hits the submit button, the `action.php` page is called. In this file you would have something like this:

Example 2-7. Printing data from our form

```
Hi <?php echo $_POST["name"]; ?>.
You are <?php echo $_POST["age"]; ?> years old.
```

A sample output of this script may be:

```
Hi Joe.
You are 22 years old.
```

It should be obvious what this does. There is nothing more to it. The `$_POST["name"]` and `$_POST["age"]` variables are automatically set for you by PHP. Earlier we used the `$_SERVER` autoglobal, now above we just introduced the $_POST autoglobal which contains all POST data. Notice how the *method* of our form is POST. If we used the method *GET* then our form information would live in the $_GET autoglobal instead. You may also use the $_REQUEST autoglobal if you don't care the source of your request data. It contains a mix of GET, POST, COOKIE and FILE data. See also the import_request_variables() function.

Using old code with new versions of PHP

Now that PHP has grown to be a popular scripting language, there are more resources out there that have listings of code you can reuse in your own scripts. For the most part the developers of the PHP language have tried to be backwards compatible, so a script written for an older version should run (ideally) without changes in a newer version of PHP, in practice some changes will usually be needed.

Two of the most important recent changes that affect old code are:

- The deprecation of the old `$HTTP_*_VARS` arrays (which need to be indicated as global when used inside a function or method). The following autoglobal arrays were introduced in PHP 4.1.0 (http://www.php.net/release_4_1_0.php). They are: `$_GET`, `$_POST`, `$_COOKIE`, `$_SERVER`, `$_ENV`, `$_REQUEST`, and `$_SESSION`. The older `$HTTP_*_VARS` arrays, such as $HTTP_POST_VARS, still exist and have since PHP 3.
- External variables are no longer registered in the global scope by default. In other words, as of PHP 4.2.0 (http://www.php.net/release_4_2_0.php) the PHP directive register_globals is *off* by default in `php.ini`. The preferred method of accessing these values is via the autoglobal arrays mentioned above. Older scripts, books, and tutorials may rely on this directive being on. If on, for example, one could use `$id` from the URL `http://www.example.com/foo.php?id=42`. Whether on or off, `$_GET['id']` is available.

For more details on these changes, see the section on predefined variables and links therein.

What's next?

With what you know now you should be able to understand most of the manual and also the various example scripts available in the example archives. You can also find other examples on the php.net websites in the links section: http://www.php.net/links.php.

Chapter 3. Installation

General Installation Considerations

Before installing first, you need to know what do you want to use PHP for. There are three main fields you can use PHP, as described in the What can PHP do? section:

- Server-side scripting
- Command line scripting
- Client-side GUI applications

For the first and most common form, you need three things: PHP itself, a web server and a web browser. You probably already have a web browser, and depending on your operating system setup, you may also have a web server (eg. Apache on Linux or IIS on Windows). You may also rent webspace at a company. This way, you don't need to set up anything on your own, only write your PHP scripts, upload it to the server you rent, and see the results in your browser. You can find a list of hosting companies at http://hosts.php.net/.

While setting up the server and PHP on your own, you have two choices for the method of connecting PHP to the server. For many servers PHP has a direct module interface (also called SAPI). These servers include Apache, Microsoft Internet Information Server, Netscape and iPlanet servers. Many other servers have support for ISAPI, the Microsoft module interface (OmniHTTPd for example). If PHP has no module support for your web server, you can always use it as a CGI processor. This means you set up your server to use the command line executable of PHP (`php.exe` on Windows) to process all PHP file requests on the server.

If you are also interested to use PHP for command line scripting (eg. write scripts autogenerating some images for you offline, or processing text files depending on some arguments you pass to them), you always need the command line executable. For more information, read the section about writing command line PHP applications. In this case, you need no server and no browser.

With PHP you can also write client side GUI applications using the PHP-GTK extension. This is a completely different approach than writing web pages, as you do not output any HTML, but manage windows and objects within them. For more information about PHP-GTK, please visit the site dedicated to this extension (http://gtk.php.net/). PHP-GTK is not included in the official PHP distribution.

From now on, this section deals with setting up PHP for web servers on Unix and Windows with server module interfaces and CGI executables.

Downloading PHP, the source code, and binary distributions for Windows can be found at http://www.php.net/. We recommend you to choose a mirror (http://www.php.net/mirrors.php) nearest to you for downloading the distributions.

Unix/HP-UX installs

This section contains notes and hints specific to installing PHP on HP-UX systems.

Example 3-1. Installation Instructions for HP-UX 10

```
From: paul_mckay@clearwater-it.co.uk
04-Jan-2001 09:49
(These tips are for PHP 4.0.4 and Apache v1.3.9)

So you want to install PHP and Apache on a HP-UX 10.20 box?

1. You need gzip, download a binary distribution from
http://hpux.connect.org.uk/ftp/hpux/Gnu/gzip-1.2.4a/gzip-1.2.4a-sd-10.20.depot.Z
uncompress the file and install using swinstall

2. You need gcc, download a binary distribution from
http://gatekeep.cs.utah.edu/ftp/hpux/Gnu/gcc-2.95.2/gcc-2.95.2-sd-10.20.depot.gz
gunzip this file and install gcc using swinstall.

3. You need the GNU binutils, you can download a binary distribution from
http://hpux.connect.org.uk/ftp/hpux/Gnu/binutils-2.9.1/binutils-2.9.1-sd-10.20.depot.gz
gunzip and install using swinstall.

4. You now need bison, you can download a binary distribution from
http://hpux.connect.org.uk/ftp/hpux/Gnu/bison-1.28/bison-1.28-sd-10.20.depot.gz
install as above.

5. You now need flex, you need to download the source from one of the
http://www.gnu.org mirrors. It is in the <filename>non-gnu</filename> directory of the ftp s
Download the file, gunzip, then tar -xvf it. Go into the newly created flex
directory and do a ./configure, then a make, and then a make install

If you have errors here, it’s probably because gcc etc. are not in your
PATH so add them to your PATH.

Right, now into the hard stuff.

6. Download the PHP and apache sources.

7. gunzip and tar -xvf them.

We need to hack a couple of files so that they can compile ok.

8. Firstly the configure file needs to be hacked because it seems to lose
track of the fact that you are a hpux machine, there will be a
better way of doing this but a cheap and cheerful hack is to put
    lt_target=hpux10.20
on line 47286 of the configure script.

9. Next, the Apache GuessOS file needs to be hacked. Under
apache_1.3.9/src/helpers change line 89 from
    "echo "hp${HPUXMACH}-hpux${HPUXVER}"; exit 0"
to:
    "echo "hp${HPUXMACH}-hp-hpux${HPUXVER}"; exit 0"

10. You cannot install PHP as a shared object under HP-UX so you must compile
it as a static, just follow the instructions at the Apache page.
```

```
11. PHP and apache should have compiled OK, but Apache won’t start. you need
to create a new user for Apache, eg www, or apache. You then change lines 252
and 253 of the conf/httpd.conf in Apache so that instead of
    User nobody
    Group nogroup
you have something like
    User www
    Group sys

This is because you can’t run Apache as nobody under hp-ux.
Apache and PHP should then work.

Hope this helps somebody,
Paul Mckay.
```

Unix/Linux installs

This section contains notes and hints specific to installing PHP on Linux distributions.

Using Packages

Many Linux distributions have some sort of package installation system, such as RPM. This can assist in setting up a standard configuration, but if you need to have a different set of features (such as a secure server, or a different database driver), you may need to build PHP and/or your webserver. If you are unfamiliar with building and compiling your own software, it is worth checking to see whether somebody has already built a packaged version of PHP with the features you need.

Unix/Mac OS X installs

This section contains notes and hints specific to installing PHP on Mac OS X Server.

Using Packages

There are a few pre-packaged and pre-compiled versions of PHP for Mac OS X. This can help in setting up a standard configuration, but if you need to have a different set of features (such as a secure server, or a different database driver), you may need to build PHP and/or your web server yourself. If you are unfamiliar with building and compiling your own software, it’s worth checking whether somebody has already built a packaged version of PHP with the features you need.

Compiling for OS X server

There are two slightly different versions of Mac OS X, client and server. The following is for OS X Server.

Example 3-2. Mac OS X server install

```
1. Get the latest distributions of Apache and PHP
2. Untar them, and run the configure program on Apache like so.
    ./configure --exec-prefix=/usr \
    --localstatedir=/var \
    --mandir=/usr/share/man \
    --libexecdir=/System/Library/Apache/Modules \
    --iconsdir=/System/Library/Apache/Icons \
    --includedir=/System/Library/Frameworks/Apache.framework/Versions/1.3/Headers \
    --enable-shared=max \
    --enable-module=most \
    --target=apache

4. You may also want to add this line:
    setenv OPTIM=-O2
    If you want the compiler to do some optimization.

5. Next, go to the PHP 4 source directory and configure it.
    ./configure --prefix=/usr \
    --sysconfdir=/etc \
    --localstatedir=/var \
    --mandir=/usr/share/man \
    --with-xml \
    --with-apache=/src/apache_1.3.12

    If you have any other additions (MySQL, GD, etc.), be sure to add
    them here. For the --with-apache string, put in the path to your
    apache source directory, for example "/src/apache_1.3.12".
6. make
7. make install
    This will add a directory to your Apache source directory under
    src/modules/php4.

8. Now, reconfigure Apache to build in PHP 4.
    ./configure --exec-prefix=/usr \
    --localstatedir=/var \
    --mandir=/usr/share/man \
    --libexecdir=/System/Library/Apache/Modules \
    --iconsdir=/System/Library/Apache/Icons \
    --includedir=/System/Library/Frameworks/Apache.framework/Versions/1.3/Headers \
    --enable-shared=max \
    --enable-module=most \
    --target=apache \
    --activate-module=src/modules/php4/libphp4.a

    You may get a message telling you that libmodphp4.a is out of date.
    If so, go to the src/modules/php4 directory inside your apache
    source directory and run this command:

    ranlib libmodphp4.a
    Then go back to the root of the apache source directory and run the
    above configure command again. That’ll bring the link table up to
    date.

9. make

10. make install

11. copy and rename the php.ini-dist file to your "bin" directory from your
    PHP 4 source directory:
    cp php.ini-dist /usr/local/bin/php.ini

    or (if your don’t have a local directory)

    cp php.ini-dist /usr/bin/php.ini
```

Other examples for Mac OS X client (http://www.stepwise.com/Articles/Workbench/Apache-1.3.14-MacOSX.html) and Mac OS X server (http://www.stepwise.com/Articles/Workbench/Apache-1.3.14-MacOSX.html) are available at Stepwise (http://www.stepwise.com/).

Compiling for MacOS X client

Those tips are graciously provided by Marc Liyanage (http://www.entropy.ch/software/macosx/).

The PHP module for the Apache web server included in Mac OS X. This version includes support for the MySQL and PostgreSQL databases.

NOTE: Be careful when you do this, you could screw up your Apache web server!

Do this to install:

- 1. Open a terminal window
- 2. Type "wget http://www.diax.ch/users/liyanage/software/macosx/libphp4.so.gz", wait for download to finish
- 3. Type "gunzip libphp4.so.gz"
- 4. Type "sudo apxs -i -a -n php4 libphp4.so"

Now type "`sudo open -a TextEdit /etc/httpd/httpd.conf`" TextEdit will open with the web server configuration file. Locate these two lines towards the end of the file: (Use the Find command)

```
#AddType application/x-httpd-php .php
#AddType application/x-httpd-php-source .phps
```

Remove the two hash marks (#), then save the file and quit TextEdit.

Finally, type "`sudo apachectl graceful`" to restart the web server.

PHP should now be up and running. You can test it by dropping a file into your "Sites" folder which is called "test.php". Into that file, write this line: "`<?php phpinfo() ?>`".

Now open up `127.0.0.1/~your_username/test.php` in your web browser. You should see a status table with information about the PHP module.

Unix/OpenBSD installs

This section contains notes and hints specific to installing PHP on OpenBSD (http://www.openbsd.org/).

Using Ports

This is the recommended method of installing PHP on OpenBSD, as it will have the latest patches and security fixes applied to it by the maintainers. To use this method, ensure that you have a recent ports tree (http://www.openbsd.org/ports.html). Then simply find out which flavors you wish to install, and issue the **make install** command. Below is an example of how to do this.

Example 3-3. OpenBSD Ports Install Example

```
$ cd /usr/ports/www/php4
$ make show VARNAME=FLAVORS
 (choose which flavors you want from the list)
$ env FLAVOR="imap gettext ldap mysql gd" make install
$ /usr/local/sbin/php4-enable
```

Using Packages

There are pre-compiled packages available for your release of OpenBSD (http://www.openbsd.org/). These integrate automatically with the version of Apache installed with the OS. However, since there are a large number of options (called *flavors*) available for this port, you may find it easier to compile it from source using the ports tree. Read the packages(7) (http://www.openbsd.org/cgi-bin/man.cgi?query=packages) manual page for more information in what packages are available.

Unix/Solaris installs

This section contains notes and hints specific to installing PHP on Solaris systems.

Required software

Solaris installs often lack C compilers and their related tools. The required software is as follows:

- gcc (recommended, other C compilers may work)
- make
- flex
- bison
- m4
- autoconf
- automake
- perl
- gzip
- tar

In addition, you will need to install (and possibly compile) any additional software specific to your configuration, such as Oracle or MySQL.

Using Packages

You can simplify the Solaris install process by using pkgadd to install most of your needed components.

Installation on UNIX systems

This section will guide you through the general configuration and installation of PHP on Unix systems. Be sure to investigate any sections specific to your platform or web server before you begin the process.

Prerequisite knowledge and software:

- Basic UNIX skills (being able to operate "make" and a C compiler, if compiling)
- An ANSI C compiler (if compiling)
- flex (for compiling)
- bison (for compiling)
- A web server
- Any module specific components (such as gd, pdf libs, etc.)

There are several ways to install PHP for the Unix platform, either with a compile and configure process, or through various pre-packaged methods. This documentation is mainly focused around the process of compiling and configuring PHP.

The initial PHP setup and configuration process is controlled by the use of the commandline options of the `configure` script. This page outlines the usage of the most common options, but there are many others to play with. Check out the Complete list of configure options for an exhaustive rundown. There are several ways to install PHP:

- As an Apache module
- As an fhttpd module
- For use with AOLServer, NSAPI, phttpd, Pi3Web, Roxen, thttpd, or Zeus.
- As a CGI executable

Apache Module Quick Reference

PHP can be compiled in a number of different ways, but one of the most popular is as an Apache module. The following is a quick installation overview.

Example 3-4. Quick Installation Instructions for PHP 4 (Apache Module Version)

```
1.  gunzip apache_1.3.x.tar.gz
2.  tar xvf apache_1.3.x.tar
3.  gunzip php-x.x.x.tar.gz
4.  tar xvf php-x.x.x.tar
5.  cd apache_1.3.x
6.  ./configure --prefix=/www
7.  cd ../php-x.x.x
8.  ./configure --with-mysql --with-apache=../apache_1.3.x --enable-track-vars
9.  make
10. make install
11. cd ../apache_1.3.x
12. ./configure --activate-module=src/modules/php4/libphp4.a
13. make
14. make install
15. cd ../php-x.x.x
16. cp php.ini-dist /usr/local/lib/php.ini
17. Edit your httpd.conf or srm.conf file and add:
      AddType application/x-httpd-php .php

18. Use your normal procedure for restarting the Apache server. (You must
    stop and restart the server, not just cause the server to reload by
    use a HUP or USR1 signal.)
```

Building

When PHP is configured, you are ready to build the CGI executable. The command **make** should take care of this. If it fails and you can't figure out why, see the Problems section.

Installation on Windows systems

This section applies to Windows 95/98/Me and Windows NT/2000/XP. Do not expect PHP to work on 16 bit platforms such as Windows 3.1. Sometimes we refer to the supported Windows platforms as Win32.

There are two main ways to install PHP for Windows: either manually or by using the InstallShield installer.

If you have Microsoft Visual Studio, you can also build PHP from the original source code.

Once you have PHP installed on your Windows system, you may also want to load various extensions for added functionality.

Windows InstallShield

The Windows PHP installer available from the downloads page at http://www.php.net/, this installs the CGI version of PHP and, for IIS, PWS, and Xitami, configures the web server as well. Also note, that while the InstallShield installer is an easy way to make PHP work, it is restricted in many aspects, as automatic setup of extensions for example is not supported.

Install your selected HTTP server on your system and make sure that it works.

Run the executable installer and follow the instructions provided by the installation wizard. Two types of installation are supported - standard, which provides sensible defaults for all the settings it can, and advanced, which asks questions as it goes along.

The installation wizard gathers enough information to set up the `php.ini` file and configure the web server to use PHP. For IIS and also PWS on NT Workstation, a list of all the nodes on the server with script map settings is displayed, and you can choose those nodes to which you wish to add the PHP script mappings.

Once the installation has completed the installer will inform you if you need to restart your system, restart the server, or just start using PHP.

Warning

Be aware, that this setup of PHP is not secure. If you would like to have a secure PHP setup, you'd better go on the manual way, and set every option carefully. This automatically working setup gives you an instantly working PHP installation, but it is not meant to be used on online servers.

Manual Installation Steps

This install guide will help you manually install and configure PHP on your Windows webserver. You need to download the zip binary distribution from the downloads page at http://www.php.net/. The original version of this guide was compiled by Bob Silva (mailto:bob_silva@mail.umesd.k12.or.us), and can be found at http://www.umesd.k12.or.us/php/win32install.html.

This guide provides manual installation support for:

- Personal Web Server 3 and 4 or newer
- Internet Information Server 3 and 4 or newer
- Apache 1.3.x
- OmniHTTPd 2.0b1 and up
- Oreilly Website Pro
- Xitami
- Netscape Enterprise Server, iPlanet

PHP 4 for Windows comes in two flavours - a CGI executable (php.exe), and several SAPI modules (for example: php4isapi.dll). The latter form is new to PHP 4, and provides significantly improved performance and some new functionality.

Warning

The SAPI modules have been significantly improved in the 4.1 release, however, you may find that you encounter possible server errors or other server modules such as ASP failing, in older systems.

If you choose one of the SAPI modules and use Windows 95, be sure to download the DCOM update from the Microsoft DCOM pages (http://download.microsoft.com/msdownload/dcom/95/x86/en/dcom95.exe). For the ISAPI module, an ISAPI 4.0 compliant Web server is required (tested on IIS 4.0, PWS 4.0 and IIS 5.0). IIS 3.0 is *NOT* supported. You should download and install the Windows NT 4.0 Option Pack with IIS 4.0 if you want native PHP support.

The following steps should be performed on all installations before the server specific instructions.

- Extract the distribution file to a directory of your choice. `c:\php\` is a good start. You probably do not want to use a path in which spaces are included (for example: c:\program files\php is not a good idea). Some web servers will crash if you do.
- You need to ensure that the DLLs which PHP uses can be found. The precise DLLs involved depend on which web server you use and whether you want to run PHP as a CGI or as a server module. `php4ts.dll` is always used. If you are using a server module (e.g. ISAPI or Apache) then you will need the relevant DLL from the `sapi` folder. If you are using any PHP extension DLLs then you will need those as well. To make sure that the DLLs can be found, you can either copy them to the system directory (e.g. `winnt/system32` or `windows/system`) or you can make sure that they live in the same directory as the main PHP executable or DLL your web server will use (e.g. `php.exe`, `php4apache.dll`).

The PHP binary, the SAPI modules, and some extensions rely on external DLLs for execution. Make sure that these DLLs in the distribution exist in a directory that is in the Windows PATH. The best bet to do it is to copy the files below into your system directory, which is typically:
`c:\windows\system` for Windows 9x/ME
`c:\winnt\system32` for Windows NT/2000
`c:\windows\system32` for Windows XP
The files to copy are:
`php4ts.dll`, if it already exists there, overwrite it
The files in your distribution's 'dlls' directory. If you have them already installed on your system, overwrite them only if

Download the latest version of the Microsoft Data Access Components (MDAC) for your platform, especially if you use Microsoft Windows 9x/NT4. MDAC is available at http://www.microsoft.com/data/.

- Copy your chosen ini file (see below) to your '%WINDOWS%' directory on Windows 9x/Me or to your '%SYSTEMROOT%' directory under Windows NT/2000/XP and rename it to `php.ini`. Your '%WINDOWS%' or '%SYSTEMROOT%' directory is typically:
  `c:\windows` for Windows 9x/ME/XP
  `c:\winnt` or `c:\winnt40` for NT/2000 servers

  There are two ini files distributed in the zip file, `php.ini-dist` and `php.ini-optimized`. We advise you to use `php.ini-optimized`, because we optimized the default settings in this file for performance, and security. The best is to study all the ini settings and set every element manually yourself. If you would like to achieve the best security, then this is the way for you, although PHP works fine with these default ini files.

- Edit your new `php.ini` file:
  - You will need to change the 'extension_dir' setting to point to your php-install-dir, or where you have placed your `php_*.dll` files. ex: `c:\php\extensions`
  - If you are using OmniHTTPd, do not follow the next step. Set the 'doc_root' to point to your webservers document_root. For example: `c:\apache\htdocs` or `c:\webroot`
  - Choose which extensions you would like to load when PHP starts. See the section about Windows extensions, about how to set up one, and what is already built in. Note that on a new installation it is advisable to first get PHP working and tested without any extensions before enabling them in `php.ini`.
  - On PWS and IIS, you can set the `browscap.ini` to point to: `c:\windows\system\inetsrv\browscap.ini` on Windows 9x/Me, `c:\winnt\system32\inetsrv\browscap.ini` on NT/2000, and `c:\windows\system32\inetsrv\browscap.ini` on XP.
  - Note that the `mibs` directory supplied with the Windows distribution contains support files for SNMP. This directory should be moved to `DRIVE:\usr\mibs` (`DRIVE` being the drive where PHP is installed.)
  - If you're using NTFS on Windows NT, 2000 or XP, make sure that the user running the webserver has read permissions to your `php.ini` (e.g. make it readable by Everyone).
- For PWS give execution permission to the webroot:

- Start PWS Web Manager
- Edit Properties of the "Home"-Directory
- Select the "execute"-Checkbox

Building from source

Before getting started, it is worthwhile answering the question: "Why is building on Windows so hard?" Two reasons come to mind:

1. Windows does not (yet) enjoy a large community of developers who are willing to freely share their source. As a direct result, the necessary investment in infrastructure required to support such development hasn't been made. By and large, what is available has been made possible by the porting of necessary utilities from Unix. Don't be surprised if some of this heritage shows through from time to time.
2. Pretty much all of the instructions that follow are of the "set and forget" variety. So sit back and try follow the instructions below as faithfully as you can.

Preparations

Before you get started, you have a lot to download...

- For starters, get the Cygwin toolkit from the closest cygwin (http://sources.redhat.com/cygwin/download.html) mirror site. This will provide you most of the popular GNU utilities used by the build process.
- Download the rest of the build tools you will need from the PHP site at http://www.php.net/extra/win32build.zip.
- Get the source code for the DNS name resolver used by PHP at http://www.php.net/extra/bindlib_w32.zip. This is a replacement for the `resolv.lib` library included in `win32build.zip`.
- If you don't already have an unzip utility, you will need one. A free version is available from InfoZip (http://www.cdrom.com/pub/infozip/UnZip.html).
- If you plan to compile PHP as a static Apache module you will also need the Apache sources (http://httpd.apache.org/dist/httpd/) of your version of Apache.

Finally, you are going to need the source to PHP 4 itself. You can get the latest development version using anonymous CVS (http://www.php.net/anoncvs.php). If you get a snapshot (http://snaps.php.net/) or a source (http://www.php.net/downloads.php) tarball, you not only will have to untar and ungzip it, but you will have to convert the bare linefeeds to crlf's in the `*.dsp` and `*.dsw` files before Microsoft Visual C++ will have anything to do with them.

> **Note:** Place the `Zend` and `TSRM` directories inside the `php4` directory in order for the projects to be found during the build process.

Putting it all together

- Follow the instructions for installing the unzip utility of your choosing.
- Execute `setup.exe` and follow the installation instructions. If you choose to install to a path other than `c:\cygnus`, let the build process know by setting the Cygwin environment variable. On Windows 95/98 setting an environment variable can be done by placing a line in your `autoexec.bat`. On Windows NT, go to My Computer => Control Panel => System and select the environment tab.

  Warning

  Make a temporary directory for Cygwin to use, otherwise many commands (particularly bison) will fail. On Windows 95/98, **`mkdir C:\TMP`**. For Windows NT, **`mkdir %SystemDrive%\tmp`**.

- Make a directory and unzip `win32build.zip` into it.
- Launch Microsoft Visual C++, and from the menu select Tools => Options. In the dialog, select the directories tab. Sequentially change the dropdown to Executables, Includes, and Library files, and ensure that `cygwin\bin`, `win32build\include`, and `win32build\lib` are in each list, respectively. (To add an entry, select a blank line at the end of the list and begin typing). Typical entries will look like this:
  - `c:\cygnus\bin`
  - `c:\php-win32build\include`
  - `c:\php-win32build\lib`

  Press OK, and exit out of Visual C++.

- Make another directory and unzip `bindlib_w32.zip` into it. Decide whether you want to have debug symbols available (bindlib - Win32 Debug) or not (bindlib - Win32 Release). Build the appropriate configuration:
  - For GUI users, launch VC++, and then select File => Open Workspace and select bindlib. Then select Build=>Set Active Configuration and select the desired configuration. Finally select Build=>Rebuild All.
  - For command line users, make sure that you either have the C++ environment variables registered, or have run **vcvars.bat**, and then execute one of the following:
    - **`msdev bindlib.dsp /MAKE "bindlib - Win32 Debug"`**
    - **`msdev bindlib.dsp /MAKE "bindlib - Win32 Release"`**

  - At this point, you should have a usable `resolv.lib` in either your `Debug` or `Release` subdirectories. Copy this file into your `win32build\lib` directory over the file by the same name found in there.

Compiling

The best way to get started is to build the standalone/CGI version.

- For GUI users, launch VC++, and then select File => Open Workspace and select php4ts. Then select Build=>Set Active Configuration and select the desired configuration. Finally select Build=>Rebuild All.
- For command line users, make sure that you either have the C++ environment variables registered, or have run **vcvars.bat**, and then execute one of the following:
  - **`msdev php4ts.dsp /MAKE "php4ts - Win32 Debug_TS"`**
  - **`msdev php4ts.dsp /MAKE "php4ts - Win32 Release_TS"`**
  - At this point, you should have a usable `php.exe` in either your `Debug_TS` or `Release_TS` subdirectories.

Repeat the above steps with `php4isapi.dsp` (which can be found in `sapi\isapi`) in order to build the code necessary for integrating PHP with Microsoft IIS.

It is possible to do minor customization to the build process by editing the main/config.win32.h.in file.

Installation of Windows extensions

After installing PHP and a webserver on Windows, you will probably want to install some extensions for added functionality. The following table describes some of the extensions available. You can choose which extensions you would like to load when PHP starts by uncommenting the: ’extension=php_*.dll’ lines in `php.ini`. You can also load a module dynamically in your script using dl().

The DLLs for PHP extensions are prefixed with ’php_’ in PHP 4 (and ’php3_’ in PHP 3). This prevents confusion between PHP extensions and their supporting libraries.

> **Note:** In PHP 4.0.6 BCMath, Calendar, COM, FTP, MySQL, ODBC, PCRE, Session, WDDX and XML support is *built in*. You don’t need to load any additional extensions in order to use these functions. See your distributions `README.txt` or `install.txt` for a list of built in modules.

> **Note:** Some of these extensions need extra DLLs to work. Couple of them can be found in the distribution package, in the ’dlls’ folder but some, for example Oracle (php_oci8.dll) require DLLs which are not bundled with the distribution package.
>
> Copy the bundled DLLs from ’DLLs’ folder to your Windows PATH, safe places are:
>
> c:\windows\system for Windows 9x/Me
> c:\winnt\system32 for Windows NT/2000
> c:\windows\system32 for Windows XP
>
> If you have them already installed on your system, overwrite them only if something doesn’t work correctly (Before overwriting them, it is a good idea to make a backup of them, or move them to another folder - just in case something goes wrong).

Table 3-1. PHP Extensions

Extension	Description	Notes
php_bz2.dll	bzip2 compression functions	None
php_calendar.dll	Calendar conversion functions	Built in since PHP 4.0.3
php_cpdf.dll	ClibPDF functions	None
php_crack.dll	Crack functions	None
php3_crypt.dll	Crypt functions	unknown
php_ctype.dll	ctype family functions	None
php_curl.dll	CURL, Client URL library functions	Requires: libeay32.dll, ssleay32.dll (bundled)
php_cybercash.dll	Cybercash payment functions	None
php_db.dll	DBM functions	Deprecated. Use DBA instead (php_dba.dll)
php_dba.dll	DBA: DataBase (dbm-style) Abstraction layer functions	None
php_dbase.dll	dBase functions	None
php3_dbm.dll	Berkeley DB2 library	unknown
php_domxml.dll	DOM XML functions	Requires: libxml2.dll (bundled)
php_dotnet.dll	.NET functions	None
php_exif.dll	Read EXIF headers from JPEG	None
php_fbsql.dll	FrontBase functions	None
php_fdf.dll	FDF: Forms Data Format functions.	Requires: fdftk.dll (bundled)
php_filepro.dll	filePro functions	Read-only access
php_ftp.dll	FTP functions	Built-in since PHP 4.0.3
php_gd.dll	GD library image functions	None
php_gettext.dll	Gettext functions	Requires: gnu_gettext.dll (bundled)
php_hyperwave.dll	HyperWave functions	None
php_iconv.dll	ICONV characterset conversion	Requires: iconv-1.3.dll (bundled)
php_ifx.dll	Informix functions	Requires: Informix libraries
php_iisfunc.dll	IIS management functions	None
php_imap.dll	IMAP POP3 and NNTP functions	PHP 3: php3_imap4r1.dll
php_ingres.dll	Ingres II functions	Requires: Ingres II libraries
php_interbase.dll	InterBase functions	Requires: gds32.dll (bundled)
php_java.dll	Java extension	Requires: jvm.dll (bundled)
php_ldap.dll	LDAP functions	Requires: libsasl.dll (bundled)

Extension	Description	Notes
php_mhash.dll	Mhash Functions	None
php_ming.dll	Ming functions for Flash	None
php_msql.dll	mSQL functions	Requires: msql.dll (bundled)
php3_msql1.dll	mSQL 1 client	unknown
php3_msql2.dll	mSQL 2 client	unknown
php_mssql.dll	MSSQL functions	Requires: ntwdblib.dll (bundled)
php3_mysql.dll	MySQL functions	Built-in in PHP 4
php3_nsmail.dll	Netscape mail functions	unknown
php3_oci73.dll	Oracle functions	unknown
php_oci8.dll	Oracle 8 functions	Requires: Oracle 8 client libraries
php_openssl.dll	OpenSSL functions	Requires: libeay32.dll (bundled)
php_oracle.dll	Oracle functions	Requires: Oracle 7 client libraries
php_pdf.dll	PDF functions	None
php_pgsql.dll	PostgreSQL functions	None
php_printer.dll	Printer functions	None
php_xslt.dll	XSLT functions	Requires: sablot.dll (bundled)
php_snmp.dll	SNMP get and walk functions	NT only!
php_sybase_ct.dll	Sybase functions	Requires: Sybase client libraries
php_yaz.dll	YAZ functions	None
php_zlib.dll	ZLib compression functions	None

Servers-CGI/Commandline

The default is to build PHP as a CGI program. This creates a commandline interpreter, which can be used for CGI processing, or for non-web-related PHP scripting. If you are running a web server PHP has module support for, you should generally go for that solution for performance reasons. However, the CGI version enables Apache users to run different PHP-enabled pages under different user-ids. Please make sure you read through the Security chapter if you are going to run PHP as a CGI.

Testing

If you have built PHP as a CGI program, you may test your build by typing **make test**. It is always a good idea to test your build. This way you may catch a problem with PHP on your platform early instead of having to struggle with it later.

Benchmarking

If you have built PHP 3 as a CGI program, you may benchmark your build by typing **make bench**. Note that if Safe Mode is on by default, the benchmark may not be able to finish if it takes longer then the 30 seconds allowed. This is because the set_time_limit() can not be used in safe mode. Use the max_execution_time configuration setting to control this time for your own scripts. **make bench** ignores the configuration file.

> **Note: make bench** is only available for PHP 3.

Using Variables

Some server supplied enviroment variables are not defined in the current CGI/1.1 specification. Only the following variables are defined there; everything else should be treated as 'vendor extensions': AUTH_TYPE, CONTENT_LENGTH, CONTENT_TYPE, GATEWAY_INTERFACE, PATH_INFO, PATH_TRANSLATED, QUERY_STRING, REMOTE_ADDR, REMOTE_HOST, REMOTE_IDENT, REMOTE_USER, REQUEST_METHOD, SCRIPT_NAME, SERVER_NAME, SERVER_PORT, SERVER_PROTOCOL and SERVER_SOFTWARE

Servers-Apache

This section contains notes and hints specific to Apache installs of PHP, both for Unix and Windows versions.

Details of installing PHP with Apache on Unix

You can select arguments to add to the **configure** on line 10 below from the Complete list of configure options. The version numbers have been omitted here, to ensure the instructions are not incorrect. You will need to replace the 'xxx' here with the correct values from your files.

Example 3-5. Installation Instructions (Apache Shared Module Version) for PHP 4

```
1.  gunzip apache_xxx.tar.gz
2.  tar -xvf apache_xxx.tar
3.  gunzip php-xxx.tar.gz
4.  tar -xvf php-xxx.tar
5.  cd apache_xxx
6.  ./configure --prefix=/www --enable-module=so
7.  make
8.  make install
9.  cd ../php-xxx
10. ./configure --with-mysql --with-apxs=/www/bin/apxs
11. make
12. make install

  If you decide to change your configure options after installation
  you only need to repeat the last three steps. You only need to
  restart apache for the new module to take effect. A recompile of
  Apache is not needed.

13. cp php.ini-dist /usr/local/lib/php.ini

  You can edit your .ini file to set PHP options.  If
  you prefer this file in another location, use
  --with-config-file-path=/path in step 10.

14. Edit your httpd.conf or srm.conf file and check that these lines are
    present and not commented out:

   AddType application/x-httpd-php .php

   LoadModule php4_module        libexec/libphp4.so

  You can choose any extension you wish here.  .php is simply the one
  we suggest. You can even include .html, and .php3 can be added for
  backwards compatibility.

  The path on the right hand side of the LoadModule statement must point
  to the path of the PHP module on your system. The above statement is
  correct for the steps shown above.


15. Use your normal procedure for starting the Apache server. (You must
    stop and restart the server, not just cause the server to reload by
    use a HUP or USR1 signal.)
```

Depending on your Apache install and Unix variant, there are many possible ways to stop and restart the server. Below are some typical lines used in restarting the server, for different apache/unix installations. You should replace `/path/to/` with the path to these applications on your systems.

```
1. Several Linux and SysV variants:
/etc/rc.d/init.d/httpd restart

2. Using apachectl scripts:
/path/to/apachectl stop
/path/to/apachectl start

3. httpdctl and httpsdctl (Using OpenSSL), similar to apachectl:
/path/to/httpsdctl stop
/path/to/httpsdctl start

4. Using mod_ssl, or another SSL server, you may want to manually
stop and start:
/path/to/apachectl stop
/path/to/apachectl startssl
```

The locations of the apachectl and http(s)dctl binaries often vary. If your system has `locate` or `whereis` or `which` commands, these can assist you in finding your server control programs.

Different examples of compiling PHP for apache are as follows:

```
./configure --with-apxs --with-pgsql
```

This will create a `libphp4.so` shared library that is loaded into Apache using a LoadModule line in Apache's `httpd.conf` file. The PostgreSQL support is embedded into this `libphp4.so` library.

```
./configure --with-apxs --with-pgsql=shared
```

This will create a `libphp4.so` shared library for Apache, but it will also create a `pgsql.so` shared library that is loaded into PHP either by using the extension directive in `php.ini` file or by loading it explicitly in a script using the dl() function.

```
./configure --with-apache=/path/to/apache_source --with-pgsql
```

This will create a `libmodphp4.a` library, a `mod_php4.c` and some accompanying files and copy this into the `src/modules/php4` directory in the Apache source tree. Then you compile Apache using `--activate-module=src/modules/php4/libphp4.a` and the Apache build system will create `libphp4.a` and link it statically into the `httpd` binary. The PostgreSQL support is included directly into this `httpd` binary, so the final result here is a single `httpd` binary that includes all of Apache and all of PHP.

```
./configure --with-apache=/path/to/apache_source --with-pgsql=shared
```

Same as before, except instead of including PostgreSQL support directly into the final `httpd` you will get a `pgsql.so` shared library that you can load into PHP from either the `php.ini` file or directly using dl().

When choosing to build PHP in different ways, you should consider the advantages and drawbacks of each method. Building as a shared object will mean that you can compile apache separately, and don't have to recompile everything as you add to, or change, PHP. Building PHP into apache (static method) means that PHP will load and run faster. For more information, see the Apache webpage on DSO support (http://httpd.apache.org/docs/dso.html).

> **Note:** Apache's default http.conf currently ships with a section that looks like this:
>
> ```
> User nobody
> Group "#-1"
> ```
>
> Unless you change that to "Group nogroup" or something like that ("Group daemon" is also very common) PHP will not be able to open files.

> **Note:** Make sure you specify the installed version of apxs when using --with-apxs=/path/to/apxs. You must NOT use the apxs version that is in the apache sources but the one that is actually installed on your system.

Installing PHP on Windows with Apache 1.3.x

There are two ways to set up PHP to work with Apache 1.3.x on Windows. One is to use the CGI binary (php.exe), the other is to use the Apache module DLL. In either case you need to stop the Apache server, and edit your `srm.conf` or `httpd.conf` to configure Apache to work with PHP.

It is worth noting here that now the SAPI module has been made more stable under windows, we recommend it's use above the CGI binary, since it is more transparent and secure.

Although there can be a few variations of configuring PHP under Apache, these are simple enough to be used by the newcomer. Please consult the Apache Docs for further configuration directives.

If you unziped the PHP package to c:\php\ as described in the Manual Installation Steps section, you need to insert these lines to your Apache configuration file to set up the CGI binary:

- `ScriptAlias /php/ "c:/php/"`
- `AddType application/x-httpd-php .php .phtml`
- `Action application/x-httpd-php "/php/php.exe"`

Note that the second line in the list above can be found in the actual versions of `httpd.conf`, but it is commented out. Remember also to substitute the `c:/php/` for your actual path to PHP.

> **Warning**
>
> By using the CGI setup, your server is open to several possible attacks. Please read our CGI security section to learn how to defend yourself from attacks.

If you would like to use PHP as a module in Apache, be sure to move `php4ts.dll` to the windows/system (for Windows 9x/Me) or winnt/system32 (for Windows NT/2000/XP) directory, overwriting any older file. Then you should add the following two lines to you Apache conf file:

- `LoadModule php4_module c:/php/sapi/php4apache.dll`
- `AddType application/x-httpd-php .php .phtml`

After changing the configuration file, remember to restart the server, for example, `NET STOP APACHE` followed by `NET START APACHE`, if you run Apache as a Windows Service, or use your regular shortcuts.

> **Note:** You may find after using the windows installer for Apache that you need to define the `AddModule` directive for `mod_php4.c` in the configuration file (`httpd.conf`). This is done by adding `AddModule mod_php4.c` to the `AddModule` list, near the beginning of the configuration file. This is especially important if the `ClearModuleList` directive is defined. Failure to do this may mean PHP will not be registered as an Apache module.

There are 2 ways you can use the source code highlighting feature, however their ability to work depends on your installation. If you have configured Apache to use PHP as an ISAPI module, then by adding the following line to your configuration file you can use this feature: `AddType application/x-httpd-php-source .phps`

If you chose to configure Apache to use PHP as a CGI binary, you will need to use the show_source() function. To do this simply create a PHP script file and add this code: `<?php show_source ("original_php_script.php"); ?>`. Substitute `original_php_script.php` with the name of the file you wish to show the source of.

> **Note:** On Win-Apache all backslashes in a path statement such as "c:\directory\file.ext", must be converted to forward slashes, as "c:/directory/file.ext".

Servers-Caudium

PHP 4 can be built as a Pike module for the Caudium webserver. Note that this is not supported with PHP 3. Follow the simple instructions below to install PHP 4 for Caudium.

Example 3-6. Caudium Installation Instructions

```
1.  Make sure you have Caudium installed prior to attempting to
    install PHP 4. For PHP 4 to work correctly, you will need Pike
    7.0.268 or newer. For the sake of this example we assume that
    Caudium is installed in /opt/caudium/server/.
```

```
2.  Change directory to php-x.y.z (where x.y.z is the version number).
3.  ./configure --with-caudium=/opt/caudium/server
4.  make
5.  make install
6.  Restart Caudium if it’s currently running.
7.  Log into the graphical configuration interface and go to the
    virtual server where you want to add PHP 4 support.
8.  Click Add Module and locate and then add the PHP 4 Script Support module.
9.  If the documentation says that the ’PHP 4 interpreter isn’t
    available’, make sure that you restarted the server. If you did
    check /opt/caudium/logs/debug/default.1 for any errors related to
    <filename>PHP4.so</filename>. Also make sure that
    <filename>caudium/server/lib/[pike-version]/PHP4.so</filename>
    is present.
10. Configure the PHP Script Support module if needed.
```

You can of course compile your Caudium module with support for the various extensions available in PHP 4. See the complete list of configure options for an exhaustive rundown.

Note: When compiling PHP 4 with MySQL support you must make sure that the normal MySQL client code is used. Otherwise there might be conflicts if your Pike already has MySQL support. You do this by specifying a MySQL install directory the --with-mysql option.

Servers-fhttpd

To build PHP as an fhttpd module, answer "yes" to "Build as an fhttpd module?" (the --with-fhttpd=*DIR* option to configure) and specify the fhttpd source base directory. The default directory is `/usr/local/src/fhttpd`. If you are running fhttpd, building PHP as a module will give better performance, more control and remote execution capability.

Servers-IIS/PWS

This section contains notes and hints specific to IIS (Microsoft Internet Information Server). Installing PHP for PWS/IIS 3, PWS 4 or newer and IIS 4 or newer versions.

Windows and PWS/IIS 3

The recommended method for configuring these servers is to use the REG file included with the distribution (pws-php4cgi.reg). You may want to edit this file and make sure the extensions and PHP install directories match your configuration. Or you can follow the steps below to do it manually.

Warning

These steps involve working directly with the Windows registry. One error here can leave your system in an unstable state. We highly recommend that you back up your registry first. The PHP Development team will not be held responsible if you damage your registry.

- Run Regedit.
- Navigate to: `HKEY_LOCAL_MACHINE /System /CurrentControlSet /Services /W3Svc /Parameters /ScriptMap`.
- On the edit menu select: `New->String Value`.
- Type in the extension you wish to use for your php scripts. For example `.php`
- Double click on the new string value and enter the path to `php.exe` in the value data field. ex: `c:\php\php.exe`.
- Repeat these steps for each extension you wish to associate with PHP scripts.

The following steps do not affect the web server installation and only apply if you want your php scripts to be executed when they are run from the command line (ex. run `c:\myscripts\test.php`) or by double clicking on them in a directory viewer window. You may wish to skip these steps as you might prefer the PHP files to load into a text editor when you double click on them.

- Navigate to: `HKEY_CLASSES_ROOT`
- On the edit menu select: `New->Key`.
- Name the key to the extension you setup in the previous section. ex: `.php`
- Highlight the new key and in the right side pane, double click the "default value" and enter `phpfile`.
- Repeat the last step for each extension you set up in the previous section.
- Now create another `New->Key` under `HKEY_CLASSES_ROOT` and name it `phpfile`.
- Highlight the new key `phpfile` and in the right side pane, double click the "default value" and enter `PHP Script`.
- Right click on the `phpfile` key and select `New->Key`, name it `Shell`.
- Right click on the `Shell` key and select `New->Key`, name it `open`.
- Right click on the `open` key and select `New->Key`, name it `command`.
- Highlight the new key `command` and in the right side pane, double click the "default value" and enter the path to `php.exe`. ex: `c:\php\php.exe -q %1`. (don't forget the `%1`).
- Exit Regedit.
- If using PWS on Windows, reboot to reload the registry.

PWS and IIS 3 users now have a fully operational system. IIS 3 users can use a nifty tool (http://www.genusa.com/iis/iiscfg.html) from Steven Genusa to configure their script maps.

Windows and PWS 4 or newer

When installing PHP on Windows with PWS 4 or newer version, you have two options. One to set up the PHP CGI binary, the other is to use the ISAPI module DLL.

If you choose the CGI binary, do the following:

- Edit the enclosed `pws-php4cgi.reg` file (look into the SAPI dir) to reflect the location of your php.exe. Forward slashes should be escaped, for example: `[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\w3svc\parameters\Script Map] ".php"="c:\\php\\php.exe"`
- In the PWS Manager, right click on a given directory you want to add PHP support to, and select Properties. Check the ’Execute’ checkbox, and confirm.

If you choose the ISAPI module, do the following:

- Edit the enclosed `pws-php4isapi.reg` file (look into the SAPI dir) to reflect the location of your php4isapi.dll. Forward slashes should be escaped, for example: `[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\w3svc\parameters\Script Map] ".php"="c:\\php\\sapi\\php4isapi.dll"`
- In the PWS Manager, right click on a given directory you want to add PHP support to, and select Properties. Check the ’Execute’ checkbox, and confirm.

Windows NT/2000/XP and IIS 4 or newer

To install PHP on an NT/2000/XP Server running IIS 4 or newer, follow these instructions. You have two options to set up PHP, using the CGI binary (php.exe) or with the ISAPI module.

In either case, you need to start the Microsoft Management Console (may appear as ’Internet Services Manager’, either in your Windows NT 4.0 Option Pack branch or the Control Panel=>Administrative Tools under Windows 2000/XP). Then right click on your Web server node (this will most probably appear as ’Default Web Server’), and select ’Properties’.

If you want to use the CGI binary, do the following:

- Under ’Home Directory’, ’Virtual Directory’, or ’Directory’, click on the ’Configuration’ button, and then enter the App Mappings tab.
- Click Add, and in the Executable box, type: `c:\php\php.exe` (assuming that you have unziped PHP in c:\php\).
- In the Extension box, type the file name extension you want associated with PHP scripts. Leave ’Method exclusions’ blank, and check the Script engine checkbox. You may also like to check the

'check that file exists' box - for a small performance penalty, IIS (or PWS) will check that the script file exists and sort out authentication before firing up php. This means that you will get sensible 404 style error messages instead of cgi errors complaining that php did not output any data.

You must start over from the previous step for each extension you want associated with PHP scripts. `.php` and `.phtml` are common, although `.php3` may be required for legacy applications.

- Set up the appropriate security. (This is done in Internet Service Manager), and if your NT Server uses NTFS file system, add execute rights for I_USR_ to the directory that contains `php.exe`.

To use the ISAPI module, do the following:

- If you don't want to perform HTTP Authentication using PHP, you can (and should) skip this step. Under ISAPI Filters, add a new ISAPI filter. Use PHP as the filter name, and supply a path to the php4isapi.dll.
- Under 'Home Directory', click on the 'Configuration' button. Add a new entry to the Application Mappings. Use the path to the php4isapi.dll as the Executable, supply `.php` as the extension, leave Method exclusions blank, and check the Script engine checkbox.
- Stop IIS completely (NET STOP iisadmin)
- Start IIS again (NET START w3svc)

Servers-Netscape and iPlanet

This section contains notes and hints specific to Netscape and iPlanet installs of PHP, both for Sun Solaris and Windows versions.

You can find more information about setting up PHP for the Netscape Enterprise Server here: http://benoit.noss.free.fr/php/install-php4.html

Installing PHP with Netscape on Sun Solaris

To build PHP with NES or iPlanet web servers, enter the proper install directory for the --with-nsapi = *`DIR`* option. The default directory is usually `/opt/netscape/suitespot/`. Please also read `/php-xxx-version/sapi/nsapi/nsapi-readme.txt`.

Example 3-7. Installation Example for Netscape Enterprise on Solaris

```
Instructions for Sun Solaris 2.6 with Netscape Enterprise Server 3.6
From: bhager@invacare.com

1. Install the following packages from www.sunfreeware.com or another
download site:
    flex-2_5_4a-sol26-sparc-local
    gcc-2_95_2-sol26-sparc-local
    gzip-1.2.4-sol26-sparc-local
    perl-5_005_03-sol26-sparc-local
    bison-1_25-sol26-sparc-local
    make-3_76_1-sol26-sparc-local
    m4-1_4-sol26-sparc-local
    autoconf-2.13
    automake-1.4
    mysql-3.23.24-beta (if you want mysql support)
    tar-1.13 (GNU tar)

2. Make sure your path includes the proper directories
    PATH=.:/usr/local/bin:/usr/sbin:/usr/bin:/usr/ccs/bin
    export PATH

3. gunzip php-x.x.x.tar.gz (if you have a .gz dist, otherwise go to 4)
4. tar xvf php-x.x.x.tar
5. cd ../php-x.x.x

6. For the following step, make sure /opt/netscape/suitespot/ is where
   your netscape server is installed. Otherwise, change to correct path:
    /configure --with-mysql=/usr/local/mysql --with-nsapi=/opt/netscape/suitespot/ --enable-
7. make
8. make install
```

After performing the base install and reading the appropriate readme file, you may need to performs some additional configuration steps.

Firstly you may need to add some paths to the LD_LIBRARY_PATH environment for Netscape to find all the shared libs. This can best done in the start script for your Netscape server. Windows users can probably skip this step. The start script is often located in: `/path/to/server/https-servername/start`

You may also need to edit the configuration files that are located in:`/path/to/server/https-servername/config/`.

Example 3-8. Configuration Example for Netscape Enterprise

```
Configuration Instructions for Netscape Enterprise Server
From: bhager@invacare.com

1. Add the following line to mime.types:
    type=magnus-internal/x-httpd-php exts=php

2. Add the following to obj.conf, shlib will vary depending on
    your OS, for unix it will be something like
    /opt/netscape/suitespot/bin/libphp4.so.
    You should place the following lines after mime types init.
    Init fn="load-modules" funcs="php4_init,php4_close,php4_execute,php4_auth_trans" shlib="
    Init fn=php4_init errorString="Failed to initialize PHP!"

    <object name="default">
    .
    .
    .
    .#NOTE this next line should happen after all 'ObjectType' and before all 'AddLog' lines
    Service fn="php4_execute" type="magnus-internal/x-httpd-php"
    .
    .
    </Object>


    <Object name="x-httpd-php">
    ObjectType fn="force-type" type="magnus-internal/x-httpd-php"
    Service fn=php4_execute
    </Object>


    Authentication configuration

    PHP authentication cannot be used with any other authentication. ALL AUTHENTICATION IS
    PASSED TO YOUR PHP SCRIPT. To configure PHP Authentication for the entire server, add
    the following line:

    <Object name="default">
    AuthTrans fn=php4_auth_trans
    .
    .
    .
    .
    </Object>

    To use PHP Authentication on a single directory, add the following:

    <Object ppath="d:\path\to\authenticated\dir\*">
    AuthTrans fn=php4_auth_trans
    </Object>
```

If you are running Netscape Enterprise 4.x, then you should use the following:

Example 3-9. Configuration Example for Netscape Enterprise 4.x

```
Place these lines after the mime types init, and everything else is similar
to the example configuration above.
From: Graeme Hoose (GraemeHoose@BrightStation.com)

Init fn="load-modules" shlib="/path/to/server4/bin/libphp4.so" funcs="php4_init,php4_close,p
Init fn="php4_init" LateInit="yes"
```

Installing PHP with Netscape on Windows

To Install PHP as CGI (for Netscape Enterprise Server, iPlanet, perhaps Fastrack), do the following:

- Copy `php4ts.dll` to your systemroot (the directory where you installed windows)
- Make a file association from the command line. Type the following two lines:

  ```
  assoc .php=PHPScript
  ftype PHPScript=c:\php\php.exe %1 %*
  ```

- In the Netscape Enterprise Administration Server create a dummy shellcgi directory and remove it just after (this step creates 5 important lines in obj.conf and allow the web server to handle shellcgi scripts).
- In the Netscape Enterprise Administration Server create a new mime type (Category: type, Content-Type: magnus-internal/shellcgi, File Suffix:php).
- Do it for each web server instance you want php to run

More details about setting up PHP as a CGI executable can be found here: http://benoit.noss.free.fr/php/install-php.html

To Install PHP as NSAPI (for Netscape Enterprise Server, iPlanet, perhaps Fastrack, do the following:

- Copy `php4ts.dll` to your systemroot (the directory where you installed windows)
- Make a file association from the command line. Type the following two lines:

  ```
  assoc .php=PHPScript
  ftype PHPScript=c:\php\php.exe %1 %*
  ```

- In the Netscape Enterprise Administration Server create a new mime type (Category: type, Content-Type: magnus-internal/x-httpd-php, File Suffix:php).
- Stop your web service and edit `obj.conf`. At the end of the Init section, place these two lines (necessarily after mime type init!):

  ```
  Init fn="load-modules" funcs="php4_init,php4_close,php4_execute,php4_auth_trans" shlib="c:
  Init fn="php4_init" errorString="Failed to initialise PHP!"
  ```

- In The `< Object name="default" >` section, place this line necessarily after all 'ObjectType' and before all 'AddLog' lines:

  ```
  Service fn="php4_execute" type="magnus-internal/x-httpd-php"
  ```

- At the end of the file, create a new object called `x-httpd-php`, by inserting these lines:

  ```
  <Object name="x-httpd-php">
  ObjectType fn="force-type" type="magnus-internal/x-httpd-php"
  Service fn=php4_execute
  </Object>
  ```

- Restart your web service and apply changes
- Do it for each web server instance you want PHP to run

More details about setting up PHP as an NSAPI filter can be found here:
http://benoit.noss.free.fr/php/install-php4.html

Servers-OmniHTTPd Server

This section contains notes and hints specific to OmniHTTPd.

OmniHTTPd 2.0b1 and up for Windows

You need to complete the following steps to make PHP work with OmniHTTPd. This is a CGI executable setup. SAPI is supported by OmniHTTPd, but some tests have shown that it is not so stable to use PHP as an ISAPI module.

- Step 1: Install OmniHTTPd server.
- Step 2: Right click on the blue OmniHTTPd icon in the system tray and select `Properties`
- Step 3: Click on `Web Server Global Settings`
- Step 4: On the 'External' tab, enter: `virtual = .php | actual = c:\path-to-php-dir\php.exe`, and use the Add button.
- Step 5: On the `Mime` tab, enter: `virtual = wwwserver/stdcgi | actual = .php`, and use the Add button.
- Step 6: Click `OK`

Repeat steps 2 - 6 for each extension you want to associate with PHP.

Note: Some OmniHTTPd packages come with built in PHP support. You can choose at setup time to do a custom setup, and uncheck the PHP component. We recommend you to use the latest PHP binaries. Some OmniHTTPd servers come with PHP 4 beta distributions, so you should choose not to set up the built in support, but install your own. If the server is already on your machine, use the Replace button in Step 4 and 5 to set the new, correct information.

Servers-Oreilly Website Pro

This section contains notes and hints specific to Oreilly Website Pro.

Oreilly Website Pro 2.5 and up for Windows

This list describes how to set up the PHP CGI binary or the ISAPI module to work with Oreilly Website Pro on Windows.

- Edit the Server Properties and select the tab "Mapping".
- From the List select "Associations" and enter the desired extension (`.php`) and the path to the CGI exe (ex. `c:\php\php.exe`) or the ISAPI DLL file (ex. `c:\php\sapi\php4isapi.dll`).
- Select "Content Types" add the same extension (`.php`) and enter the content type. If you choose the CGI executable file, enter ’wwwserver/shellcgi’, if you choose the ISAPI module, enter ’wwwserver/isapi’ (both without quotes).

Servers-Sambar

This section contains notes and hints specific to the Sambar server for Windows.

Sambar Windows

This list describes how to set up the ISAPI module to work with the Sambar server on Windows.

- Find the file called mappings.ini (in the config directory) in the Sambar isntall directory.
- Open `mappings.ini` and add the following line under `[ISAPI]`:

  ```
  *.php = c:\php\php4isapi.dll
  ```

  (This line assumes that PHP was installed in `c:\php`.)

- Now restart the Sambar server for the changes to take effect.

Servers-Xitami

This section contains notes and hints specific to Xitami.

Xitami for Windows

This list describes how to set up the PHP CGI binary to work with Xitami on Windows.

- Make sure the webserver is running, and point your browser to xitamis admin console (usually `http://127.0.0.1/admin`), and click on Configuration.
- Navigate to the Filters, and put the extension which PHP should parse (i.e. .php) into the field File extensions (.xxx).
- In Filter command or script put the path and name of your php executable i.e. `c:\php\php.exe`.
- Press the ’Save’ icon.
- Restart the server to reflect changes.

Servers-Other web servers

PHP can be built to support a large number of web servers. Please see Server-related options for a full list of server-related configure options. The PHP CGI binaries are compatible with almost all webservers supporting the CGI standard.

Problems?

Read the FAQ

Some problems are more common than others. The most common ones are listed in the PHP FAQ, part of this manual.

Other problems

If you are still stuck, someone on the PHP installation mailing list may be able to help you. You should check out the archive first, in case someone already answered someone else who had the same problem as you. The archives are available from the support page on http://www.php.net/. To subscribe to the PHP installation mailing list, send an empty mail to php-install-subscribe@lists.php.net (mailto:php-install-subscribe@lists.php.net). The mailing list address is `php-install@lists.php.net`.

If you want to get help on the mailing list, please try to be precise and give the necessary details about your environment (which operating system, what PHP version, what web server, if you are running PHP as CGI or a server module, etc.), and preferably enough code to make others able to reproduce and test your problem.

Bug reports

If you think you have found a bug in PHP, please report it. The PHP developers probably don't know about it, and unless you report it, chances are it won't be fixed. You can report bugs using the bug-tracking system at http://bugs.php.net/. Please do not send bug reports in mailing list or personal letters. The bug system is also suitable to submit feature requests.

Read the How to report a bug (http://bugs.php.net/how-to-report.php) document before submitting any bug reports!

Complete list of configure options

> **Note:** These are only used at compile time. If you want to alter PHP's runtime configuration, please see the chapter on Configuration.

The following is a complete list of options supported by PHP 4 `configure` scripts (as of 4.1.0), used when compiling in Unix-like environments. Some are available in PHP 3, some in PHP 4, and some in both. This is not noted yet.

There are general configuration options for the **configure** script, consult the appropriate manual pages for GNU **autoconf** or use the command **configure --help** for a full, up-to-date list.

- Database
- Graphics
- Miscellaneous
- PHP Behaviour
- Server
- XML

Configure Options in PHP 4

> **Note:** These options are only used in PHP 4 as of PHP 4.1.0. Some are available in older versions of PHP 4, some even in PHP 3, some only in PHP 4.1.0. If you want to compile an older version, some options will probably not be available.

Database options

`--with-db`

Include old xDBM support (deprecated).

`--enable-dba=shared`

Build DBA as a shared module.

`--with-gdbm[=DIR]`

Include GDBM support.

`--with-ndbm[=DIR]`

Include NDBM support.

`--with-db2[=DIR]`

Include Berkeley DB2 support.

`--with-db3[=DIR]`

Include Berkeley DB3 support.

`--with-dbm[=DIR]`

Include DBM support.

`--with-cdb[=DIR]`

Include CDB support.

`--enable-dbase`

Enable the bundled dbase library.

`--with-dbplus`

Include dbplus support.

`--enable-dbx`

Enable dbx.

`--with-fbsql[=DIR]`

Include FrontBase support. DIR is the FrontBase base directory.

`--enable-filepro`

Enable the bundled read-only filePro support.

--with-fribidi[=DIR]

Include fribidi support (requires FriBidi >=0.1.12). DIR is the fribidi installation directory - default /usr/local/.

--with-informix[=DIR]

Include Informix support. DIR is the Informix base install directory, defaults to nothing.

--with-ingres[=DIR]

Include Ingres II support. DIR is the Ingres base directory (default /II/ingres).

--with-interbase[=DIR]

Include InterBase support. DIR is the InterBase base install directory, defaults to /usr/interbase.

--with-msql[=DIR]

Include mSQL support. DIR is the mSQL base install directory, defaults to /usr/local/Hughes.

--with-mysql[=DIR]

Include MySQL support. DIR is the MySQL base directory. If unspecified, the bundled MySQL library will be used.

--with-oci8[=DIR]

Include Oracle-oci8 support. Default DIR is ORACLE_HOME.

--with-adabas[=DIR]

Include Adabas D support. DIR is the Adabas base install directory, defaults to /usr/local.

--with-sapdb[=DIR]

Include SAP DB support. DIR is SAP DB base install directory, defaults to /usr/local.

--with-solid[=DIR]

Include Solid support. DIR is the Solid base install directory, defaults to /usr/local/solid.

--with-ibm-db2[=DIR]

Include IBM DB2 support. DIR is the DB2 base install directory, defaults to /home/db2inst1/sqllib.

--with-empress[=DIR]

Include Empress support. DIR is the Empress base install directory, defaults to $EMPRESSPATH. From PHP4, this option only supports Empress Version 8.60 and above.

--with-empress-bcs[=DIR]

Include Empress Local Access support. DIR is the Empress base install directory, defaults to $EMPRESSPATH. From PHP4, this option only supports Empress Version 8.60 and above.

--with-birdstep[=DIR]

Include Birdstep support. DIR is the Birdstep base install directory, defaults to /usr/local/birdstep.

--with-custom-odbc[=DIR]

Include a user defined ODBC support. The DIR is ODBC install base directory, which defaults to /usr/local. Make sure to define CUSTOM_ODBC_LIBS and have some odbc.h in your include dirs. E.g., you should define following for Sybase SQL Anywhere 5.5.00 on QNX, prior to run configure script: CPPFLAGS="-DODBC_QNX -DSQLANY_BUG" LDFLAGS=-lunix CUSTOM_ODBC_LIBS="-ldblib -lodbc".

--with-iodbc[=DIR]

Include iODBC support. DIR is the iODBC base install directory, defaults to /usr/local.

--with-esoob[=DIR]

Include Easysoft OOB support. DIR is the OOB base install directory, defaults to /usr/local/easysoft/oob/client.

--with-unixODBC[=DIR]

Include unixODBC support. DIR is the unixODBC base install directory, defaults to /usr/local.

--with-openlink[=DIR]

Include OpenLink ODBC support. DIR is the OpenLink base install directory, defaults to /usr/local. This is the same as iODBC.

--with-dbmaker[=DIR]

Include DBMaker support. DIR is the DBMaker base install directory, defaults to where the latest version of DBMaker is installed (such as /home/dbmaker/3.6).

--with-oracle[=DIR]

Include Oracle-oci7 support. Default DIR is ORACLE_HOME.

--with-ovrimos[=DIR]

Include Ovrimos SQL Server support. DIR is the Ovrimos' libsqlcli install directory.

--with-pgsql[=DIR]

Include PostgreSQL support. DIR is the PostgreSQL base install directory, defaults to /usr/local/pgsql. Set DIR to shared to build as a dl, or shared,DIR to build as a dl and still specify DIR.

--with-sybase[=DIR]

Include Sybase-DB support. DIR is the Sybase home directory, defaults to /home/sybase.

--with-sybase-ct[=DIR]

Include Sybase-CT support. DIR is the Sybase home directory. Defaults to /home/sybase.

--disable-unified-odbc

Disable unified ODBC support. Only applicable if iODBC, Adabas, Solid, Velocis or a custom ODBC interface is enabled. PHP 3 only!

Graphics options

--with-gd[=DIR]

Include GD support (DIR is GD's install dir). Set DIR to shared to build as a dl, or shared,DIR to build as a dl and still specify DIR.

--enable-gd-native-ttf

GD: Enable TrueType string function in gd.

--with-jpeg-dir=DIR

GD: Set the path to libjpeg install prefix.

--with-png-dir=DIR

GD: Set the path to libpng install prefix.

--with-xpm-dir=DIR

GD: Set the path to libXpm install prefix.

--with-freetype-dir=DIR

GD: Set the path to freetype2 install prefix.

--with-ttf[=DIR]

GD: Include FreeType 1.x support.

--with-t1lib[=DIR]

GD: Include T1lib support.

--with-cpdflib[=DIR]

Include cpdflib support (requires cpdflib >= 2). DIR is the cpdfllib install directory, defaults to /usr.

--with-jpeg-dir[=DIR]

jpeg dir for cpdflib 2.x.

--with-tiff-dir[=DIR]

tiff dir for cpdflib 2.x.

--with-pdflib[=DIR]

Include PDFlib support. DIR is the pdflib base install directory, defaults to /usr/local. Set DIR to shared to build as dl, or shared,DIR to build as dl and still specify DIR.

--with-jpeg-dir[=DIR]

PDFLIB: define libjpeg install directory.

--with-png-dir[=DIR]

PDFLIB: define libpng install directory.

--with-tiff-dir[=DIR]

PDFLIB: define libtiff install directory.

--with-swf[=DIR]

Include swf support.

--without-gd

Disable GD support. PHP 3 only!

--with-imagick

Include ImageMagick support. DIR is the install directory, and if left out, PHP will try to find it on its own. [experimental]. PHP 3 only!

--with-ming[=DIR]

Include ming support.

Misc options

--enable-force-cgi-redirect

Enable the security check for internal server redirects. You should use this if you are running the CGI version with Apache.

--enable-discard-path

If this is enabled, the PHP CGI binary can safely be placed outside of the web tree and people will not be able to circumvent .htaccess security.

--with-fastcgi=SRCDIR

Build PHP as FastCGI application.

--enable-debug

Compile with debugging symbols.

--with-layout=TYPE

Sets how installed files will be laid out. Type is one of PHP (default) or GNU.

--with-pear=DIR

Install PEAR in DIR (default PREFIX/lib/php).

--without-pear

Do not install PEAR.

--with-openssl[=DIR]

Include OpenSSL support (requires OpenSSL >= 0.9.5).

--enable-sigchild

Enable PHP’s own SIGCHLD handler.

--disable-rpath

Disable passing additional runtime library search paths.

--enable-libgcc

Enable explicitly linking against libgcc.

--enable-dmalloc

Enable dmalloc.

--enable-php-streams

Include experimental php streams. Do not use unless you are testing the code!

--with-zlib-dir=<DIR>

Define the location of zlib install directory.

--with-zlib[=DIR]

Include zlib support (requires zlib >= 1.0.9). DIR is the zlib install directory.

--with-aspell[=DIR]

Include ASPELL support.

--enable-bcmath

Enable bc style precision math functions.

--with-bz2[=DIR]

Include BZip2 support.

--enable-calendar

Enable support for calendar conversion.

--with-ccvs[=DIR]

Include CCVS support.

--with-crack[=DIR]

Include crack support.

--enable-ctype

Enable ctype support.

--with-curl[=DIR]

Include CURL support.

--with-cybercash[=DIR]

Include CyberCash support. DIR is the CyberCash MCK install directory.

--with-cybermut[=DIR]

Include CyberMut (French Credit Mutuel telepaiement).

--with-cyrus

Include cyrus IMAP support.

--enable-exif

Enable exif support.

--with-fdftk[=DIR]

Include fdftk support.

--enable-ftp

Enable FTP support.

--with-gettext[=DIR]

Include GNU gettext support. DIR is the gettext install directory, defaults to /usr/local.

--with-gmp

Include gmp support.

--with-hyperwave

Include Hyperwave support.

--with-icap[=DIR]

Include ICAP support.

--with-iconv[=DIR]

Include iconv support.

--with-imap[=DIR]

Include IMAP support. DIR is the c-client install prefix.

--with-kerberos[=DIR]

IMAP: Include Kerberos support. DIR is the Kerberos install dir.

--with-imap-ssl[=DIR]

IMAP: Include SSL support. DIR is the OpenSSL install dir.

--with-ircg-config

Path to the ircg-config script.

--with-ircg

Include ircg support.

--with-java[=DIR]

Include Java support. DIR is the base install directory for the JDK. This extension can only be built as a shared dl.

--with-ldap[=DIR]

Include LDAP support. DIR is the LDAP base install directory.

--enable-mailparse

Enable mailparse support.

--enable-mbstring

Enable multibyte string support.

--enable-mbstr-enc-trans

Enable japanese encoding translation.

--with-mcal[=DIR]

Include MCAL support.

--with-mcrypt[=DIR]

Include mcrypt support. DIR is the mcrypt install directory.

--with-mhash[=DIR]

Include mhash support. DIR is the mhash install directory.

--with-mnogosearch[=DIR]

Include mnoGoSearch support. DIR is the mnoGoSearch base install directory, defaults to /usr/local/mnogosearch.

--with-muscat[=DIR]

Include muscat support.

--with-ncurses

Include ncurses support.

--enable-pcntl

Enable experimental pcntl support (CGI ONLY!)

--without-pcre-regex

Do not include Perl Compatible Regular Expressions support. Use --with-pcre-regex=DIR to specify DIR where PCRE’s include and library files are located, if not using bundled library.

`--with-pfpro[=DIR]`

Include Verisign Payflow Pro support.

`--disable-posix`

Disable POSIX-like functions.

`--with-pspell[=DIR]`

Include PSPELL support.

`--with-qtdom`

Include QtDOM support (requires Qt >= 2.2.0).

`--with-libedit[=DIR]`

Include libedit readline replacement.

`--with-readline[=DIR]`

Include readline support. DIR is the readline install directory.

`--with-recode[=DIR]`

Include recode support. DIR is the recode install directory.

`--with-satellite[=DIR]`

Enable CORBA support via Satellite (EXPERIMENTAL) DIR is the base directory for ORBit.

`--with-mm[=DIR]`

Include mm support for session storage.

`--enable-trans-sid`

Enable transparent session id propagation.

`--disable-session`

Disable session support.

`--enable-shmop`

Enable shmop support.

`--with-snmp[=DIR]`

Include SNMP support. DIR is the SNMP base install directory, defaults to searching through a number of common locations for the snmp install. Set DIR to shared to build as a dl, or shared,DIR to build as a dl and still specify DIR.

`--enable-ucd-snmp-hack`

Enable UCD SNMP hack.

`--enable-sockets`

Enable sockets support.

`--with-regex=TYPE`

regex library type: system, apache, php.

`--with-system-regex`

Use system regex library (deprecated).

`--enable-sysvsem`

Enable System V semaphore support.

`--enable-sysvshm`

Enable the System V shared memory support.

`--with-vpopmail[=DIR]`

Include vpopmail support.

`--with-tsrm-pthreads`

Use POSIX threads (default).

`--enable-shared[=PKGS]`

Build shared libraries [default=yes].

`--enable-static[=PKGS]`

Build static libraries [default=yes].

`--enable-fast-install[=PKGS]`

Optimize for fast installation [default=yes].

`--with-gnu-ld`

Assume the C compiler uses GNU ld [default=no].

`--disable-libtool-lock`

Avoid locking (might break parallel builds).

`--with-pic`

Try to use only PIC/non-PIC objects [default=use both].

`--with-yaz[=DIR]`

Include YAZ support (ANSI/NISO Z39.50). DIR is the YAZ bin install directory.

`--enable-memory-limit`

Compile with memory limit support.

`--disable-url-fopen-wrapper`

Disable the URL-aware fopen wrapper that allows accessing files via HTTP or FTP.

`--enable-versioning`

Export only required symbols. See INSTALL for more information.

`--disable-bcmath`

Compile without bc style precision math functions. PHP 3 only!

`--with-imsp[=DIR]`

Include IMSp support (DIR is IMSP's include dir and libimsp.a dir). PHP 3 only!

`--with-ftp`

Include FTP support. PHP 3 only!

`--with-mck[=DIR]`

Include Cybercash MCK support. DIR is the cybercash mck build directory, defaults to /usr/src/mck-3.2.0.3-linux for help look in extra/cyberlib. PHP 3 only!

`--disable-overload`

Disable user-space object overloading support. PHP 3 only!

`--enable-yp`

Include YP support. PHP 3 only!

`--with-zip`

Include ZIP support (requires zziplib >= 0.10.6). PHP 3 only!

`--with-mod-dav=DIR`

Include DAV support through Apache's mod_dav, DIR is mod_dav's installation directory (Apache module version only!) PHP 3 only!

`--enable-debugger`

Compile with remote debugging functions. PHP 3 only!

`--enable-versioning`

Take advantage of versioning and scoping provided by Solaris 2.x and Linux. PHP 3 only!

PHP options

`--enable-maintainer-mode`

Enable make rules and dependencies not useful (and sometimes confusing) to the casual installer.

`--with-config-file-path=PATH`

Sets the path in which to look for `php.ini`, defaults to PREFIX/lib.

`--enable-safe-mode`

Enable safe mode by default.

`--with-exec-dir[=DIR]`

Only allow executables in DIR when in safe mode defaults to /usr/local/php/bin.

`--enable-magic-quotes`

Enable magic quotes by default.

`--disable-short-tags`

Disable the short-form <? start tag by default.

Server options

`--with-aolserver=DIR`

Specify path to the installed AOLserver.

`--with-apxs[=FILE]`

Build shared Apache module. FILE is the optional pathname to the Apache apxs tool; defaults to apxs. Make sure you specify the version of apxs that is actually installed on your system and NOT the one that is in the apache source tarball.

`--with-apache[=DIR]`

Build Apache module. DIR is the top-level Apache build directory, defaults to /usr/local/apache.

`--with-mod_charset`

Enable transfer tables for mod_charset (Rus Apache).

`--with-apxs2[=FILE]`

Build shared Apache 2.0 module. FILE is the optional pathname to the Apache apxs tool; defaults to apxs.

`--with-fhttpd[=DIR]`

Build fhttpd module. DIR is the fhttpd sources directory, defaults to /usr/local/src/fhttpd.

`--with-isapi=DIR`

Build PHP as an ISAPI module for use with Zeus.

`--with-nsapi=DIR`

Specify path to the installed Netscape Server.

`--with-phttpd=DIR`

No information yet.

`--with-pi3web=DIR`

Build PHP as a module for use with Pi3Web.

`--with-roxen=DIR`

Build PHP as a Pike module. DIR is the base Roxen directory, normally /usr/local/roxen/server.

`--enable-roxen-zts`

Build the Roxen module using Zend Thread Safety.

`--with-servlet[=DIR]`

Include servlet support. DIR is the base install directory for the JSDK. This SAPI prereqs the java extension must be built as a shared dl.

`--with-thttpd=SRCDIR`

Build PHP as thttpd module.

`--with-tux=MODULEDIR`

Build PHP as a TUX module (Linux only).

XML options

`--with-dom[=DIR]`

Include DOM support (requires libxml >= 2.4.2). DIR is the libxml install directory, defaults to /usr.

`--disable-xml`

Disable XML support using bundled expat lib.

`--with-expat-dir=DIR`

XML: external libexpat install dir.

`--with-xmlrpc[=DIR]`

Include XMLRPC-EPI support.

`--enable-wddx`

Enable WDDX support.

Chapter 4. Configuration

The configuration file

The configuration file (called `php3.ini` in PHP 3.0, and simply `php.ini` as of PHP 4.0) is read when PHP starts up. For the server module versions of PHP, this happens only once when the web server is started. For the CGI and CLI version, it happens on every invocation.

The default location of `php.ini` is a compile time option (see the FAQ entry), but can be changed for the CGI and CLI version with the `-c` command line switch, see the chapter about using `PHP` from the command line. You can also use the environment variable `PHPRC` for an additional path to search for a `php.ini` file.

Not every PHP directive is documented below. For a list of all directives, please read your well commented `php.ini` file. You may want to view the latest php.ini here (http://cvs.php.net/co.php/php4/php.ini-dist) from CVS.

> **Note:** The default value for the PHP directive register_globals changed from *on* to *off* in PHP 4.2.0 (http://www.php.net/release_4_2_0.php).

Example 4-1. `php.ini` example

```
; any text on a line after an unquoted semicolon (;) is ignored
[php] ; section markers (text within square brackets) are also ignored
; Boolean values can be set to either:
;    true, on, yes
; or false, off, no, none
register_globals = off
magic_quotes_gpc = yes

; you can enclose strings in double-quotes
include_path = ".:/usr/local/lib/php"

; backslashes are treated the same as any other character
include_path = ".;c:\php\lib"
```

How to change configuration settings

Running `PHP` as Apache module

When using PHP as an Apache module, you can also change the configuration settings using directives in Apache configuration files (e.g. `httpd.conf`) and `.htaccess` files (You will need "AllowOverride Options" or "AllowOverride All" privileges)

With PHP 3.0, there are Apache directives that correspond to each configuration setting in the `php3.ini` name, except the name is prefixed by "php3_".

With PHP 4.0, there are several Apache directives that allow you to change the PHP configuration from within the Apache configuration file itself.

php_value *`name value`*

This sets the value of the specified variable.

php_flag *`name on|off`*

This is used to set a Boolean configuration option.

php_admin_value *`name value`*

This sets the value of the specified variable. "Admin" configuration settings can only be set from within the main Apache configuration files, and not from `.htaccess` files.

php_admin_flag *`name on|off`*

This is used to set a Boolean configuration option.

Example 4-2. Apache configuration example

```
<IfModule mod_php4.c>
  php_value include_path ".:/usr/local/lib/php"
  php_admin_flag safe_mode on
</IfModule>
<IfModule mod_php3.c>
  php3_include_path ".:/usr/local/lib/php"
  php3_safe_mode on
</IfModule>
```

Note: PHP constants do not exist outside of PHP. For example, in `httpd.conf` do not use PHP constants such as `E_ALL` or `E_NOTICE` to set the error_reporting directive as they will have no meaning and will evaluate to *0*. Use the associated bitmask values instead. These constants can be used in `php.ini`

Other interfaces to PHP

Regardless of the interface to PHP you can change certain values at runtime of your scripts through ini_set(). The following table provides an overview at which level a directive can be set/changed.

Table 4-1. Definition of PHP_INI_* constants

Constant	Value	Meaning
PHP_INI_USER	1	Entry can be set in user scripts
PHP_INI_PERDIR	2	Entry can be set in `.htaccess` and VHost directives in httpd.conf.
PHP_INI_SYSTEM	4	Entry can be set in `php.ini` or `httpd.conf` (but not in VHost blocks in httpd.conf).
PHP_INI_ALL	7	Entry can be set anywhere

You can view the settings of the configuration values in the output of phpinfo(). You can also access the values of individual configuration settings using ini_get() or get_cfg_var().

Configuration directives

Httpd Options

Table 4-2. Httpd Options

Name	Default	Changeable
async_send	"0"	PHP_INI_ALL

Language Options

Table 4-3. Language and Misc Configuration Options

Name	Default	Changeable
short_open_tag	On	PHP_INI_SYSTEM|PHP_INI_PERDIR
asp_tags	Off	PHP_INI_SYSTEM|PHP_INI_PERDIR
precision	"14"	PHP_INI_ALL
y2k_compliance	Off	PHP_INI_ALL
allow_call_time_pass_reference	On	PHP_INI_SYSTEM|PHP_INI_PERDIR
expose_php	On	PHP_INI_SYSTEM

Here is a short explanation of the configuration directives.

`short_open_tag` boolean

Tells whether the short form (**`<? ?>`**) of PHP's open tag should be allowed. If you want to use PHP in combination with XML, you can disable this option in order to use **`<?xml ?>`** inline. Otherwise, you can print it with PHP, for example: **`<?php echo '<?xml version="1.0"'; ?>`**. Also if disabled, you must use the long form of the PHP open tag (**`<?php ?>`**).

> **Note:** This directive also affects the shorthand **`<?=`**, which is identical to **`<? echo`**. Use of this shortcut requires short_open_tag to be on.

`asp_tags` boolean

Enables the use of ASP-like <% %> tags in addition to the usual <?php ?> tags. This includes the variable-value printing shorthand of <%= $value %>. For more information, see Escaping from HTML.

> **Note:** Support for ASP-style tags was added in 3.0.4.

`precision` integer

The number of significant digits displayed in floating point numbers.

`y2k_compliance` boolean

Enforce year 2000 compliance (will cause problems with non-compliant browsers)

`allow_call_time_pass_reference` boolean

Whether to enable the ability to force arguments to be passed by reference at function call time. This method is deprecated and is likely to be unsupported in future versions of PHP/Zend. The encouraged method of specifying which arguments should be passed by reference is in the function declaration. You're encouraged to try and turn this option Off and make sure your scripts work properly with it in order to ensure they will work with future versions of the language (you will receive a warning each time you use this feature, and the argument will be passed by value instead of by reference).

See also References Explained.

expose_php boolean

Decides whether PHP may expose the fact that it is installed on the server (e.g. by adding its signature to the Web server header). It is no security threat in any way, but it makes it possible to determine whether you use PHP on your server or not.

Resource Limits

Table 4-4. Resource Limits

Name	Default	Changeable
memory_limit	"8M"	PHP_INI_ALL

Here is a short explanation of the configuration directives.

memory_limit integer

This sets the maximum amount of memory in bytes that a script is allowed to allocate. This helps prevent poorly written scripts for eating up all available memory on a server.

See also: max_execution_time.

Data Handling

Table 4-5. Data Handling Configuration Options

Name	Default	Changeable
track-vars	"On"	PHP_INI_??
arg_separator.output	"&"	PHP_INI_ALL
arg_separator.input	"&"	PHP_INI_SYSTEM|PHP_INI_PERDIR
variables_order	"EGPCS"	PHP_INI_ALL
register_globals	"Off"	PHP_INI_PERDIR|PHP_INI_SYSTEM
register_argc_argv	"On"	PHP_INI_PERDIR|PHP_INI_SYSTEM
post_max_size	"8M"	PHP_INI_SYSTEM|PHP_INI_PERDIR
gpc_order	"GPC"	PHP_INI_ALL

Name	Default	Changeable
auto_prepend_file	""	PHP_INI_SYSTEM|PHP_INI_PERDIR
auto_append_file	""	PHP_INI_SYSTEM|PHP_INI_PERDIR
default_mimetype	"text/html"	PHP_INI_ALL
default_charset	"iso-8859-1"	PHP_INI_ALL
always_populate_raw_post_data	"0"	PHP_INI_SYSTEM|PHP_INI_PERDIR
allow_webdav_methods	"0"	PHP_INI_SYSTEM|PHP_INI_PERDIR

Here is a short explanation of the configuration directives.

`track_vars` boolean

If enabled, then Environment, GET, POST, Cookie, and Server variables can be found in the global associative arrays `$_ENV`, `$_GET`, `$_POST`, `$_COOKIE`, and `$_SERVER`.

Note that as of PHP 4.0.3, track_vars is always turned on.

`arg_separator.output` string

The separator used in PHP generated URLs to separate arguments.

`arg_separator.input` string

List of separator(s) used by PHP to parse input URLs into variables.

Note: Every character in this directive is considered as separator!

`variables_order` string

Set the order of the EGPCS (Environment, GET, POST, Cookie, Server) variable parsing. The default setting of this directive is "EGPCS". Setting this to "GP", for example, will cause PHP to completely ignore environment variables, cookies and server variables, and to overwrite any GET method variables with POST-method variables of the same name.

See also register_globals.

`register_globals` boolean

Tells whether or not to register the EGPCS (Environment, GET, POST, Cookie, Server) variables as global variables. For example; if register_globals = on, the url `http://www.example.com/test.php?id=3` will produce `$id`. Or, `$DOCUMENT_ROOT` from `$_SERVER['DOCUMENT_ROOT']`. You may want to turn this off if you don't want to clutter your scripts' global scope with user data. As of PHP 4.2.0 (http://www.php.net/release_4_2_0.php), this directive defaults to *off*. It's preferred to go through PHP Predefined Variables instead, such as the superglobals: `$_ENV`, `$_GET`, `$_POST`, `$_COOKIE`, and `$_SERVER`. Please read the security chapter on Using register_globals for related information.

Please note that register_globals cannot be set at runtime (ini_set()). Although, you can use `.htaccess` if your host allows it as described above. An example `.htaccess` entry: **`php_flag register_globals on`**.

> **Note:** register_globals is affected by the variables_order directive.

`register_argc_argv` boolean

Tells PHP whether to declare the argv & argc variables (that would contain the GET information).

See also command line. Also, this directive became available in PHP 4.0.0 and was always "on" before that.

`post_max_size` integer

Sets max size of post data allowed. This setting also affects file upload. To upload large files, this value must be larger than upload_max_filesize.

If memory limit is enabled by your configure script, memory_limit also affects file uploading. Generally speaking, memory_limit should be larger than *`post_max_size`*.

`gpc_order` string

Set the order of GET/POST/COOKIE variable parsing. The default setting of this directive is "GPC". Setting this to "GP", for example, will cause PHP to completely ignore cookies and to overwrite any GET method variables with POST-method variables of the same name.

> **Note:** This option is not available in PHP 4. Use variables_order instead.

`auto_prepend_file` string

Specifies the name of a file that is automatically parsed before the main file. The file is included as if it was called with the include() function, so include_path is used.

The special value none disables auto-prepending.

`auto_append_file` string

Specifies the name of a file that is automatically parsed after the main file. The file is included as if it was called with the include() function, so include_path is used.

The special value none disables auto-appending.

> **Note:** If the script is terminated with exit(), auto-append will *not* occur.

`default_mimetype` string

`default_charset` string

As of 4.0b4, PHP always outputs a character encoding by default in the Content-type: header. To disable sending of the charset, simply set it to be empty.

`always_populate_raw_post_data` boolean

Always populate the $HTTP_RAW_POST_DATA variable.

`allow_webdav_methods` boolean

Allow handling of WebDAV http requests within PHP scripts (eg. PROPFIND, PROPPATCH, MOVE, COPY, etc..) If you want to get the post data of those requests, you have to set always_populate_raw_post_data as well.

See also: magic_quotes_gpc, magic-quotes-runtime, and magic_quotes_sybase.

Paths and Directories

Table 4-6. Paths and Directories Configuration Options

Name	Default	Changeable
include_path	PHP_INCLUDE_PATH	PHP_INI_ALL
doc_root	PHP_INCLUDE_PATH	PHP_INI_SYSTEM
user_dir	NULL	PHP_INI_SYSTEM
extension_dir	PHP_EXTENSION_DIR	PHP_INI_SYSTEM
cgi.force_redirect	"1"	PHP_INI_SYSTEM
cgi.redirect_status_env	""	PHP_INI_SYSTEM
fastcgi.impersonate	"0"	PHP_INI_SYSTEM

Here is a short explanation of the configuration directives.

`include_path` string

Specifies a list of directories where the require(), include() and **fopen_with_path()** functions look for files. The format is like the system's PATH environment variable: a list of directories separated with a colon in UNIX or semicolon in Windows.

Example 4-3. UNIX include_path

```
include_path=.:/home/httpd/php-lib
```

Example 4-4. Windows include_path

```
include_path=".;c:\www\phplib"
```

The default value for this directive is `.` (only the current directory).

`doc_root` string

PHP's "root directory" on the server. Only used if non-empty. If PHP is configured with safe mode, no files outside this directory are served. If PHP was not compiled with FORCE_REDIRECT, you SHOULD set doc_root if you are running php as a CGI under any web server (other than IIS) The alternative is to use the cgi.force_redirect configuration below.

`user_dir` string

The base name of the directory used on a user's home directory for `PHP` files, for example `public_html`.

`extension_dir` string

In what directory PHP should look for dynamically loadable extensions. See also: enable_dl, and dl().

`extension` string

Which dynamically loadable extensions to load when PHP starts up.

`cgi.force_redirect` boolean

cgi.force_redirect is necessary to provide security running PHP as a CGI under most web servers. Left undefined, PHP turns this on by default. You can turn it off *AT YOUR OWN RISK*.

> **Note:** Windows Users: You CAN safely turn this off for IIS, in fact, you MUST. To get OmniHTTPD or Xitami to work you MUST turn it off.

`cgi.cgi.redirect_status_env` string

If cgi.force_redirect is turned on, and you are not running under Apache or Netscape (iPlanet) web servers, you MAY need to set an environment variable name that PHP will look for to know it is OK to continue execution.

> **Note:** Setting this variable MAY cause security issues, KNOW WHAT YOU ARE DOING FIRST.

`fastcgi.impersonate` string

FastCGI under IIS (on WINNT based OS) supports the ability to impersonate security tokens of the calling client. This allows IIS to define the security context that the request runs under. mod_fastcgi under Apache does not currently support this feature (03/17/2002) Set to 1 if running under IIS. Default is zero.

File Uploads

Table 4-7. File Uploads Configuration Options

Name	Default	Changeable
file_uploads	"1"	PHP_INI_SYSTEM
upload_tmp_dir	NULL	PHP_INI_SYSTEM
upload_max_filesize	"2M"	PHP_INI_SYSTEM|PHP_INI_PERDIR

Here is a short explanation of the configuration directives.

`file_uploads` boolean

Whether or not to allow HTTP file uploads. See also the upload_max_filesize, upload_tmp_dir, and post_max_size directives.

`upload_tmp_dir` string

The temporary directory used for storing files when doing file upload. Must be writable by whatever user `PHP` is running as. If not specified PHP will use the system's default.

`upload_max_filesize` integer

The maximum size of an uploaded file.

General SQL

Table 4-8. General SQL Configuration Options

Name	Default	Changeable
sql.safe_mode	"0"	PHP_INI_SYSTEM

Here is a short explanation of the configuration directives.

`sql.safe_mode` boolean

Debugger Configuration Directives

`debugger.host` string

DNS name or IP address of host used by the debugger.

`debugger.port` string

Port number used by the debugger.

`debugger.enabled` boolean

Whether the debugger is enabled.

Chapter 5. Security

PHP is a powerful language and the interpreter, whether included in a web server as a module or executed as a separate CGI binary, is able to access files, execute commands and open network connections on the server. These properties make anything run on a web server insecure by default. PHP is designed specifically to be a more secure language for writing CGI programs than Perl or C, and with correct selection of compile-time and runtime configuration options, and proper coding practices, it can give you exactly the combination of freedom and security you need.

As there are many different ways of utilizing PHP, there are many configuration options controlling its behaviour. A large selection of options guarantees you can use PHP for a lot of purposes, but it also means there are combinations of these options and server configurations that result in an insecure setup.

The configuration flexibility of PHP is equally rivalled by the code flexibility. PHP can be used to build complete server applications, with all the power of a shell user, or it can be used for simple server-side includes with little risk in a tightly controlled environment. How you build that environment, and how secure it is, is largely up to the PHP developer.

This chapter starts with some general security advice, explains the different configuration option combinations and the situations they can be safely used, and describes different considerations in coding for different levels of security.

General considerations

A completely secure system is a virtual impossibility, so an approach often used in the security profession is one of balancing risk and usability. If every variable submitted by a user required two forms of biometric validation (such as a retinal scan and a fingerprint), you would have an extremely high level of accountability. It would also take half an hour to fill out a fairly complex form, which would tend to encourage users to find ways of bypassing the security.

The best security is often inobtrusive enough to suit the requirements without the user being prevented from accomplishing their work, or over-burdening the code author with excessive complexity. Indeed, some security attacks are merely exploits of this kind of overly built security, which tends to erode over time.

A phrase worth remembering: A system is only as good as the weakest link in a chain. If all transactions are heavily logged based on time, location, transaction type, etc. but the user is only verified based on a single cookie, the validity of tying the users to the transaction log is severely weakened.

When testing, keep in mind that you will not be able to test all possibilities for even the simplest of pages. The input you may expect will be completely unrelated to the input given by a disgruntled employee, a cracker with months of time on their hands, or a housecat walking across the keyboard. This is why it's best to look at the code from a logical perspective, to discern where unexpected data can be introduced, and then follow how it is modified, reduced, or amplified.

The Internet is filled with people trying to make a name for themselves by breaking your code, crashing your site, posting inappropriate content, and otherwise making your day interesting. It doesn't matter if you have a small or large site, you are a target by simply being online, by having a server that can be connected to. Many cracking programs do not discern by size, they simply trawl massive IP blocks looking for victims. Try not to become one.

Installed as CGI binary

Possible attacks

Using PHP as a CGI binary is an option for setups that for some reason do not wish to integrate PHP as a module into server software (like Apache), or will use PHP with different kinds of CGI wrappers to create safe chroot and setuid environments for scripts. This setup usually involves installing executable PHP binary to the web server cgi-bin directory. CERT advisory CA-96.11 (http://www.cert.org/advisories/CA-96.11.interpreters_in_cgi_bin_dir.html) recommends against placing any interpreters into cgi-bin. Even if the PHP binary can be used as a standalone interpreter, PHP is designed to prevent the attacks this setup makes possible:

- Accessing system files: `http://my.host/cgi-bin/php?/etc/passwd`

  The query information in a url after the question mark (?) is passed as command line arguments to the interpreter by the CGI interface. Usually interpreters open and execute the file specified as the first argument on the command line.

  When invoked as a CGI binary, PHP refuses to interpret the command line arguments.

- Accessing any web document on server: `http://my.host/cgi-bin/php/secret/doc.html`

  The path information part of the url after the PHP binary name, `/secret/doc.html` is conventionally used to specify the name of the file to be opened and interpreted by the CGI program. Usually some web server configuration directives (Apache: Action) are used to redirect requests to documents like `http://my.host/secret/script.php` to the PHP interpreter. With this setup, the web server first checks the access permissions to the directory `/secret`, and after that creates the redirected request `http://my.host/cgi-bin/php/secret/script.php`. Unfortunately, if the request is originally given in this form, no access checks are made by web server for file `/secret/script.php`, but only for the `/cgi-bin/php` file. This way any user able to access `/cgi-bin/php` is able to access any protected document on the web server.

  In PHP, compile-time configuration option --enable-force-cgi-redirect and runtime configuration directives doc_root and user_dir can be used to prevent this attack, if the server document tree has any directories with access restrictions. See below for full the explanation of the different combinations.

Case 1: only public files served

If your server does not have any content that is not restricted by password or ip based access control, there is no need for these configuration options. If your web server does not allow you to do redirects, or the server does not have a way to communicate to the PHP binary that the request is a safely redirected request, you can specify the option --enable-force-cgi-redirect to the configure script. You still have to make sure your PHP scripts do not rely on one or another way of calling the script, neither by directly `http://my.host/cgi-bin/php/dir/script.php` nor by redirection `http://my.host/dir/script.php`.

Redirection can be configured in Apache by using AddHandler and Action directives (see below).

Case 2: using --enable-force-cgi-redirect

This compile-time option prevents anyone from calling PHP directly with a url like `http://my.host/cgi-bin/php/secretdir/script.php`. Instead, PHP will only parse in this mode if it has gone through a web server redirect rule.

Usually the redirection in the Apache configuration is done with the following directives:

```
Action php-script /cgi-bin/php
AddHandler php-script .php
```

This option has only been tested with the Apache web server, and relies on Apache to set the non-standard CGI environment variable REDIRECT_STATUS on redirected requests. If your web server does not support any way of telling if the request is direct or redirected, you cannot use this option and you must use one of the other ways of running the CGI version documented here.

Case 3: setting doc_root or user_dir

To include active content, like scripts and executables, in the web server document directories is sometimes consider an insecure practice. If, because of some configuration mistake, the scripts are not executed but displayed as regular HTML documents, this may result in leakage of intellectual property or security information like passwords. Therefore many sysadmins will prefer setting up another directory structure for scripts that are accessible only through the PHP CGI, and therefore always interpreted and not displayed as such.

Also if the method for making sure the requests are not redirected, as described in the previous section, is not available, it is necessary to set up a script doc_root that is different from web document root.

You can set the PHP script document root by the configuration directive doc_root in the configuration file, or you can set the environment variable PHP_DOCUMENT_ROOT. If it is set, the CGI version of PHP will always construct the file name to open with this *`doc_root`* and the path information in the request, so you can be sure no script is executed outside this directory (except for *`user_dir`* below).

Another option usable here is user_dir. When user_dir is unset, only thing controlling the opened file name is *`doc_root`*. Opening an url like `http://my.host/~user/doc.php` does not result in opening a file under users home directory, but a file called `~user/doc.php` under doc_root (yes, a directory name starting with a tilde [~]).

If user_dir is set to for example `public_php`, a request like `http://my.host/~user/doc.php` will open a file called `doc.php` under the directory named `public_php` under the home directory of the user. If the home of the user is `/home/user`, the file executed is `/home/user/public_php/doc.php`.

`user_dir` expansion happens regardless of the *`doc_root`* setting, so you can control the document root and user directory access separately.

Case 4: PHP parser outside of web tree

A very secure option is to put the PHP parser binary somewhere outside of the web tree of files. In `/usr/local/bin`, for example. The only real downside to this option is that you will now have to put a line similar to:

```
#!/usr/local/bin/php
```

as the first line of any file containing PHP tags. You will also need to make the file executable. That is, treat it exactly as you would treat any other CGI script written in Perl or sh or any other common scripting language which uses the `#!` shell-escape mechanism for launching itself.

To get PHP to handle PATH_INFO and PATH_TRANSLATED information correctly with this setup, the php parser should be compiled with the --enable-discard-path configure option.

Installed as an Apache module

When PHP is used as an Apache module it inherits Apache’s user permissions (typically those of the "nobody" user). This has several impacts on security and authorization. For example, if you are using PHP to access a database, unless that database has built-in access control, you will have to make the database accessable to the "nobody" user. This means a malicious script could access and modify the database, even without a username and password. It’s entirely possible that a web spider could stumble across a database administrator’s web page, and drop all of your databases. You can protect against this with Apache authorization, or you can design your own access model using LDAP, .htaccess files, etc. and include that code as part of your PHP scripts.

Often, once security is established to the point where the PHP user (in this case, the apache user) has very little risk attached to it, it is discovered that PHP is now prevented from writing any files to user directories. Or perhaps it has been prevented from accessing or changing databases. It has equally been secured from writing good and bad files, or entering good and bad database transactions.

A frequent security mistake made at this point is to allow apache root permissions, or to escalate apache’s abilitites in some other way.

Escalating the Apache user’s permissions to root is extremely dangerous and may compromise the entire system, so sudo’ing, chroot’ing, or otherwise running as root should not be considered by those who are not security professionals.

There are some simpler solutions. By using open_basedir you can control and restrict what directories are allowed to be used for PHP. You can also set up apache-only areas, to restrict all web based activity to non-user, or non-system, files.

Filesystem Security

PHP is subject to the security built into most server systems with respect to permissions on a file and directory basis. This allows you to control which files in the filesystem may be read. Care should be taken with any files which are world readable to ensure that they are safe for reading by all users who have access to that filesystem.

Since PHP was designed to allow user level access to the filesystem, it’s entirely possible to write a PHP script that will allow you to read system files such as /etc/passwd, modify your ethernet connections, send massive printer jobs out, etc. This has some obvious implications, in that you need to ensure that the files that you read from and write to are the appropriate ones.

Consider the following script, where a user indicates that they’d like to delete a file in their home directory. This assumes a situation where a PHP web interface is regularly used for file management, so the Apache user is allowed to delete files in the user home directories.

Example 5-1. Poor variable checking leads to....

```
<?php
// remove a file from the user’s home directory
$username = $_POST[’user_submitted_name’];
$homedir = "/home/$username";
$file_to_delete = "$userfile";
unlink ($homedir/$userfile);
echo "$file_to_delete has been deleted!";
?>
```

Since the username is postable from a user form, they can submit a username and file belonging to someone else, and delete files. In this case, you’d want to use some other form of authentication. Consider what could happen if the variables submitted were "../etc/" and "passwd". The code would then effectively read:

Example 5-2. ... A filesystem attack

```
<?php
// removes a file from anywhere on the hard drive that
// the PHP user has access to. If PHP has root access:
$username = "../etc/";
$homedir = "/home/../etc/";
$file_to_delete = "passwd";
unlink ("/home/../etc/passwd");
echo "/home/../etc/passwd has been deleted!";
?>
```

There are two important measures you should take to prevent these issues.

- Only allow limited permissions to the PHP web user binary.
- Check all variables which are submitted.

Here is an improved script:

Example 5-3. More secure file name checking

```
<?php
// removes a file from the hard drive that
// the PHP user has access to.
$username = $_SERVER['REMOTE_USER']; // using an authentication mechanisim

$homedir = "/home/$username";

$file_to_delete = basename("$userfile"); // strip paths
unlink ($homedir/$file_to_delete);

$fp = fopen("/home/logging/filedelete.log","+a"); //log the deletion
$logstring = "$username $homedir $file_to_delete";
fputs ($fp, $logstring);
fclose($fp);

echo "$file_to_delete has been deleted!";
?>
```

However, even this is not without it's flaws. If your authentication system allowed users to create their own user logins, and a user chose the login "../etc/", the system is once again exposed. For this reason, you may prefer to write a more customized check:

Example 5-4. More secure file name checking

```
<?php
$username = $_SERVER['REMOTE_USER']; // using an authentication mechanisim
$homedir = "/home/$username";

if (!ereg('^[^./][^/]*$', $userfile))
     die('bad filename'); //die, do not process

if (!ereg('^[^./][^/]*$', $username))
     die('bad username'); //die, do not process
//etc...
?>
```

Depending on your operating system, there are a wide variety of files which you should be concerned about, including device entries (/dev/ or COM1), configuration files (/etc/ files and the .ini files), well known file storage areas (/home/, My Documents), etc. For this reason, it's usually easier to create a policy where you forbid everything except for what you explicitly allow.

Database Security

Nowadays, databases are cardinal components of any web based application by enabling websites to provide varying dynamic content. Since very sensitive or secret informations can be stored in such database, you should strongly consider to protect them somehow.

To retrieve or to store any information you need to connect to the database, send a legitimate query, fetch the result, and close the connecion. Nowadays, the commonly used query language in this interaction is the Structured Query Language (SQL). See how an attacker can tamper with an SQL query.

As you can realize, PHP cannot protect your database by itself. The following sections aim to be an introduction into the very basics of how to access and manipulate databases within PHP scripts.

Keep in mind this simple rule: defence in depth. In the more place you take the more action to increase the protection of your database, the less probability of that an attacker succeeds, and exposes or abuse any stored secret information. Good design of the database schema and the application deals with your greatest fears.

Designing Databases

The first step is always to create the database, unless you want to use an existing third party's one. When a database is created, it is assigned to an owner, who executed the creation statement. Usually, only the owner (or a superuser) can do anything with the objects in that database, and in order to allow other users to use it, privileges must be granted.

Applications should never connect to the database as its owner or a superuser, because these users can execute any query at will, for example, modifying the schema (e.g. dropping tables) or deleting its entire content.

You may create different database users for every aspect of your application with very limited rights to database objects. The most required privileges should be granted only, and avoid that the same user can interact with the database in different use cases. This means that if intruders gain access to your database using one of these credentials, they can only effect as many changes as your application can.

You are encouraged not to implement all the business logic in the web application (i.e. your script), instead to do it in the database schema using views, triggers or rules. If the system evolves, new ports will be intended to open to the database, and you have to reimplement the logic in each separate database client. Over and above, triggers can be used to transparently and automatically handle fields, which often provides insight when debugging problems with your application or tracing back transactions.

Connecting to Database

You may want to estabilish the connections over SSL to encrypt client/server communications for increased security, or you can use ssh to encrypt the network connection between clients and the database server. If either of them is done, then monitoring your traffic and gaining informations in this way will be a hard work.

Encrypted Storage Model

SSL/SSH protects data travelling from the client to the server, SSL/SSH does not protect the persistent data stored in a database. SSL is an on-the-wire protocol.

Once an attacker gains access to your database directly (bypassing the webserver), the stored sensitive data may be exposed or misused, unless the information is protected by the database itself. Encrypting the data is a good way to mitigate this threat, but very few databases offer this type of data encryption.

The easiest way to work around this problem is to first create your own encryption package, and then use it from within your PHP scripts. PHP can assist you in this case with its several extensions, such as Mcrypt and Mhash, covering a wide variety of encryption algorithms. The script encrypts the data be stored first, and decrypts it when retrieving. See the references for further examples how encryption works.

In case of truly hidden data, if its raw representation is not needed (i.e. not be displayed), hashing may be also taken into consideration. The well-known example for the hashing is storing the MD5 hash of a password in a database, instead of the password itself. See also crypt() and md5().

Example 5-5. Using hashed password field

```
// storing password hash
$query  = sprintf("INSERT INTO users(name,pwd) VALUES(’%s’,’%s’);",
            addslashes($username), md5($password));
$result = pg_exec($connection, $query);

// querying if user submitted the right password
$query = sprintf("SELECT 1 FROM users WHERE name=’%s’ AND pwd=’%s’;",
            addslashes($username), md5($password));
$result = pg_exec($connection, $query);

if (pg_numrows($result) > 0) {
    echo "Welcome, $username!";
}
else {
    echo "Authentication failed for $username.";
}
```

SQL Injection

Many web developers are unaware of how SQL queries can be tampered with, and assume that an SQL query is a trusted command. It means that SQL queries are able to circumvent access controls, thereby bypassing standard authentication and authorization checks, and sometimes SQL queries even may allow access to host operating system level commands.

Direct SQL Command Injection is a technique where an attacker creates or alters existing SQL commands to expose hidden data, or to override valuable ones, or even to execute dangerous system level commands on the database host. This is accomplished by the application taking user input and combining it with static parameters to build a SQL query. The following examples are based on true stories, unfortunately.

Owing to the lack of input validation and connecting to the database on behalf of a superuser or the one who can create users, the attacker may create a superuser in your database.

Example 5-6. Splitting the result set into pages ... and making superusers (PostgreSQL and MySQL)

```
$offset = argv[0]; // beware, no input validation!
$query  = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET $offset;";
// with PostgreSQL
$result = pg_exec($conn, $query);
// with MySQL
$result = mysql_query($query);
```

Normal users click on the 'next', 'prev' links where the `$offset` is encoded into the URL. The script expects that the incoming `$offset` is decimal number. However, someone tries to break in with appending urlencode()'d form of the following to the URL

```
// in case of PostgreSQL
0;
insert into pg_shadow(usename,usesysid,usesuper,usecatupd,passwd)
    select 'crack', usesysid, 't','t','crack'
    from pg_shadow where usename='postgres';
--

// in case of MySQL
0;
UPDATE user SET Password=PASSWORD('crack') WHERE user='root';
FLUSH PRIVILEGES;
```

If it happened, then the script would present a superuser access to him. Note that `0;` is to supply a valid offset to the original query and to terminate it.

> **Note:** It is common technique to force the SQL parser to ignore the rest of the query written by the developer with `--` which is the comment sign in SQL.

A feasible way to gain passwords is to circumvent your search result pages. What the attacker needs only is to try if there is any submitted variable used in SQL statement which is not handled properly. These filters can be set commonly in a preceding form to customize `WHERE, ORDER BY, LIMIT` and `OFFSET` clauses in `SELECT` statements. If your database supports the `UNION` construct, the attacker may try to append an entire query to the original one to list passwords from an arbitrary table. Using encrypted password fields is strongly encouraged.

Example 5-7. Listing out articles ... and some passwords (any database server)

```
$query  = "SELECT id, name, inserted, size FROM products
                  WHERE size = '$size'
                  ORDER BY $order LIMIT $limit, $offset;";
$result = odbc_exec($conn, $query);
```

The static part of the query can be combined with another `SELECT` statement which reveals all passwords:

```
'
union select '1', concat(uname||'-'||passwd) as name, '1971-01-01', '0' from usertable;
--
```

If this query (playing with the `'` and `--`) were assigned to one of the variables used in `$query`, the query beast awakened.

SQL UPDATEs are also subject to attacking your database. These queries are also threatened by chopping and appending an entirely new query to it. But the attacker might fiddle with the `SET` clause. In this case some schema information must be possessed to manipulate the query successfully. This can be acquired by examing the form variable names, or just simply brute forcing. There are not so many naming convention for fields storing passwords or usernames.

Example 5-8. From resetting a password ... to gaining more privileges (any database server)

```
$query = "UPDATE usertable SET pwd='$pwd' WHERE uid='$uid';";
```

But a malicious user sumbits the value `' or uid like'%admin%'; --` to `$uid` to change the admin's password, or simply sets `$pwd` to `"hehehe', admin='yes', trusted=100 "` (with a trailing space) to gain more privileges. Then, the query will be twisted:

```
// $uid == ' or uid like'%admin%'; --
$query = "UPDATE usertable SET pwd='...' WHERE uid='' or uid like '%admin%'; --";

// $pwd == "hehehe', admin='yes', trusted=100 "
$query = "UPDATE usertable SET pwd='hehehe', admin='yes', trusted=100 WHERE ...;"
```

A frightening example how operating system level commands can be accessed on some database hosts.

Example 5-9. Attacking the database host's operating system (MSSQL Server)

```
$query  = "SELECT * FROM products WHERE id LIKE '%$prod%'";
$result = mssql_query($query);
```

If attacker submits the value `a%' exec master..xp_cmdshell 'net user test testpass /ADD' --` to `$prod`, then the `$query` will be:

```
$query  = "SELECT * FROM products
                    WHERE id LIKE '%a%'
                    exec master..xp_cmdshell 'net user test testpass /ADD'--";
$result = mssql_query($query);
```

MSSQL Server executes the SQL statements in the batch including a command to add a new user to the local accounts database. If this application were running as `sa` and the MSSQLSERVER service is running with sufficient privileges, the attacker would now have an account with which to access this machine.

> **Note:** Some of the examples above is tied to a specific database server. This does not mean that a similar attack is impossible against other products. Your database server may be so vulnerable in other manner.

Avoiding techniques

You may plead that the attacker must possess a piece of information about the database schema in most examples. You are right, but you never know when and how it can be taken out, and if it happens, your database may be exposed. If you are using an open source, or publicly available database handling package, which may belong to a content management system or forum, the intruders easily produce a copy of a piece of your code. It may be also a security risk if it is a poorly designed one.

These attacks are mainly based on exploiting the code not being written with security in mind. Never trust on any kind of input, especially which comes from the client side, even though it comes from a select box, a hidden input field or a cookie. The first example shows that such a blameless query can cause disasters.

- Never connect to the database as a superuser or as the database owner. Use always customized users with very limited privileges.
- Check if the given input has the expected data type. PHP has a wide range of input validating functions, from the simplest ones found in Variable Functions and in Character Type Functions (e.g. is_numeric(), ctype_digit() respectively) onwards the Perl compatible Regular Expressions support.
- If the application waits for numerical input, consider to verify data with is_numeric(), or silently change its type using settype(), or use its numeric representation by sprintf().

Example 5-10. A more secure way to compose a query for paging

```
settype($offset, ’integer’);
$query = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET $offset;";

// please note %d in the format string, using %s would be meaningless
$query = sprintf("SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET %d;",
                 $offset);
```

- Quote each non numeric user input which is passed to the database with addslashes() or addcslashes(). See the first example. As the examples shows, quotes burnt into the static part of the query is not enough, and can be easily hacked.
- Do not print out any database specific information, especially about the schema, by fair means or foul. See also Error Reporting and Error Handling and Logging Functions.
- You may use stored procedures and previously defined cursors to abstract data access so that users do not directly access tables or views, but this solution has another impacts.

Besides these, you benefit from logging queries either within your script or by the database itself, if it supports. Obviously, the logging is unable to prevent any harmful attempt, but it can be helpful to trace back which application has been circumvented. The log is not useful by itself, but through the information it contains. The more detail is generally better.

Error Reporting

With PHP security, there are two sides to error reporting. One is beneficial to increasing security, the other is detrimental.

A standard attack tactic involves profiling a system by feeding it improper data, and checking for the kinds, and contexts, of the errors which are returned. This allows the system cracker to probe for information about the server, to determine possible weaknesses. For example, if an attacker had gleaned information about a page based on a prior form submission, they may attempt to override variables, or modify them:

Example 5-11. Attacking Variables with a custom HTML page

```
<form method="post" action="attacktarget?username=badfoo&password=badfoo">
<input type="hidden" name="username" value="badfoo">
<input type="hidden" name="password" value="badfoo">
</form>
```

The PHP errors which are normally returned can be quite helpful to a developer who is trying to debug a script, indicating such things as the function or file that failed, the PHP file it failed in, and the line number which the failure occured in. This is all information that can be exploited. It is not uncommon for a php developer to use show_source(), highlight_string(), or highlight_file() as a debugging measure, but in a live site, this can expose hidden variables, unchecked syntax, and other dangerous information. Especially dangerous is running code from known sources with built-in debugging handlers, or using common debugging techniques. If the attacker can determine what general technique you are using, they may try to brute-force a page, by sending various common debugging strings:

Example 5-12. Exploiting common debugging variables

```
<form method="post" action="attacktarget?errors=Y&amp;showerrors=1"&debug=1">
<input type="hidden" name="errors" value="Y">
<input type="hidden" name="showerrors" value="1">
<input type="hidden" name="debug" value="1">
</form>
```

Regardless of the method of error handling, the ability to probe a system for errors leads to providing an attacker with more information.

For example, the very style of a generic PHP error indicates a system is running PHP. If the attacker was looking at an .html page, and wanted to probe for the back-end (to look for known weaknesses in the system), by feeding it the wrong data they may be able to determine that a system was built with PHP.

A function error can indicate whether a system may be running a specific database engine, or give clues as to how a web page or programmed or designed. This allows for deeper investigation into open database ports, or to look for specific bugs or weaknesses in a web page. By feeding different pieces of bad data, for example, an attacker can determine the order of authentication in a script, (from the line number errors) as well as probe for exploits that may be exploited in different locations in the script.

A filesystem or general PHP error can indicate what permissions the webserver has, as well as the structure and organization of files on the web server. Developer written error code can aggravate this problem, leading to easy exploitation of formerly "hidden" information.

There are three major solutions to this issue. The first is to scrutinize all functions, and attempt to compensate for the bulk of the errors. The second is to disable error reporting entirely on the running code. The third is to use PHP’s custom error handling functions to create your own error handler. Depending on your security policy, you may find all three to be applicable to your situation.

One way of catching this issue ahead of time is to make use of PHP’s own error_reporting(), to help you secure your code and find variable usage that may be dangerous. By testing your code, prior to deployment, with E_ALL, you can quickly find areas where your variables may be open to poisoning or modification in other ways. Once you are ready for deployment, by using E_NONE, you insulate your code from probing.

Example 5-13. Finding dangerous variables with E_ALL

```
<?php
if ($username) {  // Not initialized or checked before usage
    $good_login = 1;
}
if ($good_login == 1) { // If above test fails, not initialized or checked before usage
    fpassthru ("/highly/sensitive/data/index.html");
}
?>
```

Using Register Globals

One feature of PHP that can be used to enhance security is configuring PHP with register_globals = off. By turning off the ability for any user-submitted variable to be injected into PHP code, you can reduce the amount of variable poisoning a potential attacker may inflict. They will have to take the additional time to forge submissions, and your internal variables are effectively isolated from user submitted data.

While it does slightly increase the amount of effort required to work with PHP, it has been argued that the benefits far outweigh the effort.

Example 5-14. Working with register_globals=on

```
<?php
if ($username) {  // can be forged by a user in get/post/cookies
    $good_login = 1;
}

if ($good_login == 1) { // can be forged by a user in get/post/cookies,
    fpassthru ("/highly/sensitive/data/index.html");
}
?>
```

Example 5-15. Working with register_globals = off

```
<?php
if($_COOKIE['username']){
    // can only come from a cookie, forged or otherwise
    $good_login = 1;
    fpassthru ("/highly/sensitive/data/index.html");
}
?>
```

By using this wisely, it's even possible to take preventative measures to warn when forging is being attempted. If you know ahead of time exactly where a variable should be coming from, you can check to see if submitted data is coming from an inappropriate kind of submission. While it doesn't guarantee that data has not been forged, it does require an attacker to guess the right kind of forging.

Example 5-16. Detecting simple variable poisoning

```
<?php
if ($_COOKIE['username'] &&
    !$_POST['username'] &&
    !$_GET['username'] ) {
    // Perform other checks to validate the user name...
    $good_login = 1;
    fpassthru ("/highly/sensitive/data/index.html");
} else {
   mail("admin@example.com", "Possible breakin attempt", $_SERVER['REMOTE_ADDR']);
   echo "Security violation, admin has been alerted.";
   exit;
}
?>
```

Of course, simply turning off register_globals does not mean code is secure. For every piece of data that is submitted, it should also be checked in other ways.

User Submitted Data

The greatest weakness in many PHP programs is not inherent in the language itself, but merely an issue of code not being written with security in mind. For this reason, you should always take the time to consider the implications of a given piece of code, to ascertain the possible damage if an unexpected variable is submitted to it.

Example 5-17. Dangerous Variable Usage

```
<?php
// remove a file from the user's home directory... or maybe
// somebody else's?
unlink ($evil_var);

// Write logging of their access... or maybe an /etc/passwd entry?
fputs ($fp, $evil_var);

// Execute something trivial.. or rm -rf *?
system ($evil_var);
exec ($evil_var);

?>
```

You should always carefully examine your code to make sure that any variables being submitted from a web browser are being properly checked, and ask yourself the following questions:

- Will this script only affect the intended files?
- Can unusual or undesirable data be acted upon?
- Can this script be used in unintended ways?
- Can this be used in conjunction with other scripts in a negative manner?
- Will any transactions be adequately logged?

By adequately asking these questions while writing the script, rather than later, you prevent an unfortunate re-write when you need to increase your security. By starting out with this mindset, you won't guarantee the security of your system, but you can help improve it.

You may also want to consider turning off register_globals, magic_quotes, or other convenience settings which may confuse you as to the validity, source, or value of a given variable. Working with PHP in error_reporting(E_ALL) mode can also help warn you about variables being used before they are checked or initialized (so you can prevent unusual data from being operated upon).

Hiding PHP

In general, security by obscurity is one of the weakest forms of security. But in some cases, every little bit of extra security is desirable.

A few simple techniques can help to hide PHP, possibly slowing down an attacker who is attempting to discover weaknesses in your system. By setting expose_php = off in your `php.ini` file, you reduce the amount of information available to them.

Another tactic is to configure web servers such as apache to parse different filetypes through PHP, either with an .htaccess directive, or in the apache configuration file itself. You can then use misleading file extensions:

Example 5-18. Hiding PHP as another language

```
# Make PHP code look like other code types
AddType application/x-httpd-php .asp .py .pl
```

Or obscure it completely:

Example 5-19. Using unknown types for PHP extensions

```
# Make PHP code look like unknown types
AddType application/x-httpd-php .bop .foo .133t
```

Or hide it as html code, which has a slight performance hit because all html will be parsed through the PHP engine:

Example 5-20. Using html types for PHP extensions

```
# Make all PHP code look like html
AddType application/x-httpd-php .htm .html
```

For this to work effectively, you must rename your PHP files with the above extensions. While it is a form of security through obscurity, it’s a minor preventative measure with few drawbacks.

Keeping Current

PHP, like any other large system, is under constant scrutiny and improvement. Each new version will often include both major and minor changes to enhance and repair security flaws, configuration mishaps, and other issues that will affect the overall security and stability of your system.

Like other system-level scripting languages and programs, the best approach is to update often, and maintain awareness of the latest versions and their changes.

Part II. Language Reference

Chapter 6. Basic syntax

Escaping from HTML

When PHP parses a file, it simply passes the text of the file through until it encounters one of the special tags which tell it to start interpreting the text as PHP code. The parser then executes all the code it finds, up until it runs into a PHP closing tag, which tells the parser to just start passing the text through again. This is the mechanism which allows you to embed PHP code inside HTML: everything outside the PHP tags is left utterly alone, while everything inside is parsed as code.

There are four sets of tags which can be used to denote blocks of PHP code. Of these, only two (<?php. . .?> and <script language="php">. . .</script>) are always available; the others can be turned on or off from the `php.ini` configuration file. While the short-form tags and ASP-style tags may be convenient, they are not as portable as the longer versions. Also, if you intend to embed PHP code in XML or XHTML, you will need to use the <?php. . .?> form to conform to the XML.

The tags supported by PHP are:

Example 6-1. Ways of escaping from HTML

```
1.  <?php echo("if you want to serve XHTML or XML documents, do like this\n"); ?>

2.  <? echo ("this is the simplest, an SGML processing instruction\n"); ?>
    <?= expression ?> This is a shortcut for "<? echo expression ?>"

3.  <script language="php">
        echo ("some editors (like FrontPage) don't
              like processing instructions");
    </script>

4.  <% echo ("You may optionally use ASP-style tags"); %>
    <%= $variable; # This is a shortcut for "<% echo . . ." %>
```

The first way, <?php. . .?>, is the preferred method, as it allows the use of PHP in XML-conformant code such as XHTML.

The second way is not available always. Short tags are available only when they have been enabled. This can be done via the **short_tags()** function (PHP 3 only), by enabling the short_open_tag configuration setting in the PHP config file, or by compiling PHP with the --enable-short-tags option to **configure**. Even if it is enabled by default in php.ini-dist, use of short tags are discouraged.

The fourth way is only available if ASP-style tags have been enabled using the asp_tags configuration setting.

> **Note:** Support for ASP-style tags was added in 3.0.4.

> **Note:** Using short tags should be avoided when developing applications or libraries that are meant for redistribution, or deployment on PHP servers which are not under your control, because short tags may not be supported on the target server. For portable, redistributable code, be sure not to use short tags.

The closing tag for the block will include the immediately trailing newline if one is present. Also, the closing tag automatically implies a semicolon; you do not need to have a semicolon terminating the last line of a PHP block.

PHP allows you to use structures like this:

Example 6-2. Advanced escaping

```
<?php
if ($expression) {
    ?>
    <strong>This is true.</strong>
    <?php
} else {
    ?>
    <strong>This is false.</strong>
    <?php
}
?>
```

This works as expected, because when PHP hits the ?> closing tags, it simply starts outputting whatever it finds until it hits another opening tag. The example given here is contrived, of course, but for outputting large blocks of text, dropping out of PHP parsing mode is generally more efficient than sending all of the text through echo() or print() or somesuch.

Instruction separation

Instructions are separated the same as in C or Perl - terminate each statement with a semicolon.

The closing tag (?>) also implies the end of the statement, so the following are equivalent:

```
<?php
    echo "This is a test";
?>

<?php echo "This is a test" ?>
```

Comments

PHP supports ’C’, ’C++’ and Unix shell-style comments. For example:

```
<?php
    echo "This is a test"; // This is a one-line c++ style comment
    /* This is a multi line comment
       yet another line of comment */
    echo "This is yet another test";
    echo "One Final Test"; # This is shell-style style comment
?>
```

The "one-line" comment styles actually only comment to the end of the line or the current block of PHP code, whichever comes first.

```
<h1>This is an <?php # echo "simple";?> example.</h1>
<p>The header above will say ’This is an example’.
```

You should be careful not to nest ’C’ style comments, which can happen when commenting out large blocks.

```
<?php
 /*
    echo "This is a test"; /* This comment will cause a problem */
 */
?>
```

The one-line comment styles actually only comment to the end of the line or the current block of PHP code, whichever comes first. This means that HTML code after // ?> WILL be printed: ?> skips out of the PHP mode and returns to HTML mode, and // cannot influence that.

Chapter 7. Types

Introduction

PHP supports eight primitive types.

Four scalar types:

- boolean
- integer
- floating-point number (float)
- string

Two compound types:

- array
- object

And finally two special types:

- resource
- NULL

> **Note:** In this manual you'll often find `mixed` parameters. This pseudo-type indicates multiple possibilities for that parameter.

The type of a variable is usually not set by the programmer; rather, it is decided at runtime by PHP depending on the context in which that variable is used.

> **Note:** If you want to check out the type and value of a certain expression, use var_dump().
>
> If you simply want a human-readable representation of the type for debugging, use gettype(). To check for a certain type, do *not* use gettype(), but use the `is_*type*` functions.

If you would like to force a variable to be converted to a certain type, you may either cast the variable or use the settype() function on it.

Note that a variable may behave in different manners in certain situations, depending on what type it is at the time. For more information, see the section on Type Juggling.

Booleans

This is the easiest type. A boolean expresses a truth value. It can be either `TRUE` or `FALSE`.

> **Note:** The boolean type was introduced in PHP 4.

Syntax

To specify a boolean literal, use either the keyword `TRUE` or `FALSE`. Both are case-insensitive.

```
$foo = True; // assign the value TRUE to $foo
```

Usually you use some kind of operator which returns a boolean value, and then pass it on to a control structure.

```
// == is an operator which returns a boolean
if ($action == "show_version") {
    echo "The version is 1.23";
}

// this is not necessary:
if ($show_separators == TRUE) {
    echo "<hr>\n";
}

// because you can simply type this:
if ($show_separators) {
    echo "<hr>\n";
}
```

Converting to boolean

To explicitly convert a value to boolean, use either the `(bool)` or the `(boolean)` cast. However, in most cases you do not need to use the cast, since a value will be automatically converted if an operator, function or control structure requires a boolean argument.

See also Type Juggling.

When converting to boolean, the following values are considered `FALSE`:

- the boolean `FALSE`
- the integer 0 (zero)
- the float 0.0 (zero)
- the empty string, and the string "0"
- an array with zero elements
- an object with zero elements

- the special type NULL (including unset variables)

Every other value is considered `TRUE` (including any resource).

> **Warning**
>
> `-1` is considered `TRUE`, like any other non-zero (whether negative or positive) number!

Integers

An integer is a number of the set Z = {..., -2, -1, 0, 1, 2, ...}.

See also: Arbitrary length integers and Floating point numbers

Syntax

Integers can be specified in decimal (10-based), hexadecimal (16-based) or octal (8-based) notation, optionally preceded by a sign (- or +).

If you use the octal notation, you must precede the number with a `0` (zero), to use hexadecimal notation precede the number with `0x`.

Example 7-1. Integer literals

```
$a = 1234; # decimal number
$a = -123; # a negative number
$a = 0123; # octal number (equivalent to 83 decimal)
$a = 0x1A; # hexadecimal number (equivalent to 26 decimal)
```

The size of an integer is platform-dependent, although a maximum value of about two billion is the usual value (that’s 32 bits signed). PHP does not support unsigned integers.

Integer overflow

If you specify a number beyond the bounds of the integer type, it will be interpreted as a float instead. Also, if you perform an operation that results in a number beyond the bounds of the integer type, a float will be returned instead.

```
$large_number =  2147483647;
var_dump($large_number);
// output: int(2147483647)
```

```
$large_number =  2147483648;
var_dump($large_number);
// output: float(2147483648)

// this goes also for hexadecimal specified integers:
var_dump( 0x80000000 );
// output: float(2147483648)

$million = 1000000;
$large_number =  50000 * $million;
var_dump($large_number);
// output: float(50000000000)
```

Warning

Unfortunately, there was a bug in PHP so that this does not always work correctly when there are negative numbers involved. For example: when you do `-50000 * $million`, the result will be `-429496728`. However, when both operands are positive there is no problem.

This is solved in PHP 4.1.0.

There is no integer division operator in PHP. `1/2` yields the float `0.5`.

```
var_dump( 25/7 );
// output: float(3.5714285714286)
```

Converting to integer

To explicitly convert a value to integer, use either the `(int)` or the `(integer)` cast. However, in most cases you do not need to use the cast, since a value will be automatically converted if an operator, function or control structure requires a integer argument.

See also type-juggling.

From booleans

`FALSE` will yield `0` (zero), and `TRUE` will yield `1` (one).

From floating point numbers

When converting from float to integer, the number will be rounded *towards zero*.

If the float is beyond the boundaries of integer (usually `+/- 2.15e+9 = 2^31`), the result is undefined, since the float hasn't got enough precision to give an exact integer result. No warning, not even a notice will be issued in this case!

Warning

Never cast an unknown fraction to integer, as this can sometimes lead to unexpected results.

```
echo (int) ( (0.1+0.7) * 10 ); // echoes 7!
```

See for more information the warning about float-precision.

From strings

See String conversion

From other types

Caution

Behaviour of converting to integer is undefined for other types. Currently, the behaviour is the same as if the value was first converted to boolean. However, do *not* rely on this behaviour, as it can change without notice.

Floating point numbers

Floating point numbers (AKA "floats", "doubles" or "real numbers") can be specified using any of the following syntaxes:

```
$a = 1.234; $a = 1.2e3; $a = 7E-10;
```

The size of a float is platform-dependent, although a maximum of ~1.8e308 with a precision of roughly 14 decimal digits is a common value (that's 64 bit IEEE format).

Floating point precision

It is quite usual that simple decimal fractions like `0.1` or `0.7` cannot be converted into their internal binary counterparts without a little loss of precision. This can lead to confusing results: for example, `floor((0.1+0.7)*10)` will usually return `7` instead of the expected `8` as the result of the internal representation really being something like `7.9999999999....`

This is related to the fact that it is impossible to exactly express some fractions in decimal notation with a finite number of digits. For instance, `1/3` in decimal form becomes `0.3333333. . ..`

So never trust floating number results to the last digit and never compare floating point numbers for equality. If you really need higher precision, you should use the arbitrary precision math functions or gmp functions instead.

Strings

A string is series of characters. In PHP, a character is the same as a byte, that is, there are exactly 256 different characters possible. This also implies that PHP has no native support of Unicode.

> **Note:** It is no problem for a string to become very large. There is no practical bound to the size of strings imposed by PHP, so there is no reason at all to worry about long strings.

Syntax

A string literal can be specified in three different ways.

- single quoted
- double quoted
- heredoc syntax

Single quoted

The easiest way to specify a simple string is to enclose it in single quotes (the character `'`).

To specify a literal single quote, you will need to escape it with a backslash (`\`), like in many other languages. If a backslash needs to occur before a single quote or at the end of the string, you need to double it. Note that if you try to escape any other character, the backslash too will be printed! So usually there is no need to escape the backslash itself.

> **Note:** In PHP 3, a warning will be issued at the `E_NOTICE` level when this happens.

Note: Unlike the two other syntaxes, variables will *not* be expanded when they occur in single quoted strings.

```
echo ’this is a simple string’;
echo ’You can also have embedded newlines in strings,
like this way.’;
echo ’Arnold once said: "I\’ll be back"’;
// output: ... "I’ll be back"
echo ’Are you sure you want to delete C:\\*.*?’;
// output: ... delete C:\*.*?
echo ’Are you sure you want to delete C:\*.*?’;
// output: ... delete C:\*.*?
echo ’I am trying to include at this point: \n a newline’;
// output: ... this point: \n a newline
```

Double quoted

If the string is enclosed in double-quotes ("), PHP understands more escape sequences for special characters:

Table 7-1. Escaped characters

sequence	meaning
`\n`	linefeed (LF or 0x0A (10) in ASCII)
`\r`	carriage return (CR or 0x0D (13) in ASCII)
`\t`	horizontal tab (HT or 0x09 (9) in ASCII)
`\\`	backslash
`\$`	dollar sign
`\"`	double-quote
`\[0-7]{1,3}`	the sequence of characters matching the regular expression is a character in octal notation
`\x[0-9A-Fa-f]{1,2}`	the sequence of characters matching the regular expression is a character in hexadecimal notation

Again, if you try to escape any other character, the backslash will be printed too!

But the most important pre of double-quoted strings is the fact that variable names will be expanded. See string parsing for details.

Heredoc

Another way to delimit strings is by using heredoc syntax ("<<<"). One should provide an identifier after <<<, then the string, and then the same identifier to close the quotation.

The closing identifier *must* begin in the first column of the line. Also, the identifier used must follow the same naming rules as any other label in PHP: it must contain only alphanumeric characters and underscores, and must start with a non-digit character or underscore.

Warning

It is very important to note that the line with the closing identifier contains no other characters, except *possibly* a semicolon (`;`). That means especially that the identifier *may not be indented*, and there may not be any spaces or tabs after or before the semicolon.

Heredoc text behaves just like a double-quoted string, without the double-quotes. This means that you do not need to escape quotes in your here docs, but you can still use the escape codes listed above. Variables are expanded, but the same care must be taken when expressing complex variables inside a here doc as with strings.

Example 7-2. Heredoc string quoting example

```
<?php
$str = <<<EOD
Example of string
spanning multiple lines
using heredoc syntax.
EOD;

/* More complex example, with variables. */
class foo
{
    var $foo;
    var $bar;

    function foo()
    {
        $this->foo = ’Foo’;
        $this->bar = array(’Bar1’, ’Bar2’, ’Bar3’);
    }
}

$foo = new foo();
$name = ’MyName’;

echo <<<EOT
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should print a capital ’A’: \x41
EOT;
```

```
?>
```

Note: Heredoc support was added in PHP 4.

Variable parsing

When a string is specified in double quotes or with heredoc, variables are parsed within it.

There are two types of syntax, a simple one and a complex one. The simple syntax is the most common and convenient, it provides a way to parse a variable, an array value, or an object property.

The complex syntax was introduced in PHP 4, and can be recognised by the curly braces surrounding the expression.

Simple syntax

If a dollar sign (`$`) is encountered, the parser will greedily take as much tokens as possible to form a valid variable name. Enclose the variable name in curly braces if you want to explicitly specify the end of the name.

```
$beer = ’Heineken’;
echo "$beer’s taste is great"; // works, "’" is an invalid character for varnames
echo "He drank some $beers"; // won’t work, ’s’ is a valid character for varnames
echo "He drank some ${beer}s"; // works
```

Similarly, you can also have an array index or an object property parsed. With array indices, the closing square bracket (`]`) marks the end of the index. For object properties the same rules apply as to simple variables, though with object properties there doesn’t exist a trick like the one with variables.

```
$fruits = array( ’strawberry’ => ’red’ , ’banana’ => ’yellow’ );

// note that this works differently outside string-quotes.
echo "A banana is $fruits[banana].";

echo "This square is $square->width meters broad.";

// Won’t work. For a solution, see the complex syntax.
echo "This square is $square->width00 centimeters broad.";
```

For anything more complex, you should use the complex syntax.

Complex (curly) syntax

This isn’t called complex because the syntax is complex, but because you can include complex expressions this way.

In fact, you can include any value that is in the namespace in strings with this syntax. You simply write the expression the same way as you would outside the string, and then include it in { and }. Since you can’t escape ’{’, this syntax will only be recognised when the $ is immediately following the {. (Use "{\$" or "\{$" to get a literal "{$"). Some examples to make it clear:

```
$great = ’fantastic’;
echo "This is { $great}"; // won’t work, outputs: This is { fantastic}
echo "This is {$great}";  // works, outputs: This is fantastic
echo "This square is {$square->width}00 centimeters broad.";
echo "This works: {$arr[4][3]}";

// This is wrong for the same reason
// as $foo[bar] is wrong outside a string.
echo "This is wrong: {$arr[foo][3]}";

echo "You should do it this way: {$arr[’foo’][3]}";
echo "You can even write {$obj->values[3]->name}";
echo "This is the value of the var named $name: {${$name}}";
```

String access by character

Characters within strings may be accessed by specifying the zero-based offset of the desired character after the string in curly braces.

> **Note:** For backwards compatibility, you can still use the array-braces. However, this syntax is deprecated as of PHP 4.

Example 7-3. Some string examples

```
<?php
/* Assigning a string. */
$str = "This is a string";
```

```
/* Appending to it. */
$str = $str . " with some more text";

/* Another way to append, includes an escaped newline. */
$str .= " and a newline at the end.\n";

/* This string will end up being ’<p>Number: 9</p>’ */
$num = 9;
$str = "<p>Number: $num</p>";

/* This one will be ’<p>Number: $num</p>’ */
$num = 9;
$str = ’<p>Number: $num</p>’;

/* Get the first character of a string  */
$str = ’This is a test.’;
$first = $str{0};

/* Get the last character of a string. */
$str = ’This is still a test.’;
$last = $str{strlen($str)-1};
?>
```

Useful functions

Strings may be concatenated using the ’.’ (dot) operator. Note that the ’+’ (addition) operator will not work for this. Please see String operators for more information.

There are a lot of useful functions for string modification.

See the string functions section for general functions, the regular expression functions for advanced find&replacing (in two tastes: Perl and POSIX extended).

There are also functions for URL-strings, and functions to encrypt/decrypt strings (mcrypt and mhash).

Finally, if you still didn’t find what you’re looking for, see also the character type functions.

String conversion

When a string is evaluated as a numeric value, the resulting value and type are determined as follows.

The string will evaluate as a float if it contains any of the characters ’.’, ’e’, or ’E’. Otherwise, it will evaluate as an integer.

The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero). Valid numeric data is an optional sign, followed by one or more digits (optionally containing a decimal point), followed by an optional exponent. The exponent is an ’e’ or ’E’ followed by one or more digits.

```
$foo = 1 + "10.5";              // $foo is float (11.5)
$foo = 1 + "-1.3e3";            // $foo is float (-1299)
$foo = 1 + "bob-1.3e3";         // $foo is integer (1)
$foo = 1 + "bob3";              // $foo is integer (1)
$foo = 1 + "10 Small Pigs";     // $foo is integer (11)
$foo = 4 + "10.2 Little Piggies"; // $foo is float (14.2)
$foo = "10.0 pigs " + 1;        // $foo is float (11)
$foo = "10.0 pigs " + 1.0;      // $foo is float (11)
```

For more information on this conversion, see the Unix manual page for strtod(3).

If you would like to test any of the examples in this section, you can cut and paste the examples and insert the following line to see for yourself what's going on:

```
echo "\$foo==$foo; type is " . gettype ($foo) . "<br />\n";
```

Arrays

An array in PHP is actually an ordered map. A map is a type that maps *values* to *keys*. This type is optimized in several ways, so you can use it as a real array, or a list (vector), hashtable (which is an implementation of a map), dictionary, collection, stack, queue and probably more. Because you can have another PHP-array as a value, you can also quite easily simulate trees.

Explanation of those structures is beyond the scope of this manual, but you'll find at least one example for each of those structures. For more information about those structures, we refer you to external literature about this broad topic.

Syntax

Specifying with array()

An array can be created by the array() language-construct. It takes a certain number of comma-separated `key => value` pairs.

A `key` is either an integer or a string. If a key is the standard representation of an integer, it will be interpreted as such (i.e. `"8"` will be interpreted as `8`, while `"08"` will be interpreted as `"08"`).

A value can be anything.

If you omit a key, the maximum of the integer-indices is taken, and the new key will be that maximum + 1. As integers can be negative, this is also true for negative indices. Having e.g. the highest index being

`-6` will result in being `-5` the new key. If no integer-indices exist yet, the key will be `0` (zero). If you specify a key that already has a value assigned to it, that value will be overwritten.

Using `true` as a key will evalute to integer `1` as key. Using `false` as a key will evalute to integer `0` as key. Using `NULL` as a key will evaluate to an empty string. Using an emptry string as key will create (or overwrite) a key with an empty string and its value, it is not the same as using empty brackets.

You cannot use arrays or objects as keys. Doing so will result in a warning: `Illegal offset type`.

```
array( [key =>] value
     , ...
     )
// key is either string or nonnegative integer
// value can be anything
```

Creating/modifying with square-bracket syntax

You can also modify an existing array, by explicitly setting values.

This is done by assigning values to the array while specifying the key in brackets. You can also omit the key, add an empty pair of brackets ("`[]`") to the variable-name in that case.

```
$arr[key] = value;
$arr[] = value;
// key is either string or nonnegative integer
// value can be anything
```

If `$arr` doesn't exist yet, it will be created. So this is also an alternative way to specify an array. To change a certain value, just assign a new value to it. If you want to remove a key/value pair, you need to unset() it.

Useful functions

There are quite some useful function for working with arrays, see the array-functions section.

Note: The unset() function allows unsetting keys of an array. Be aware that the array will NOT be reindexed.

```
$a = array( 1 => ’one’, 2 => ’two’, 3 => ’three’ );
unset( $a[2] );
/* will produce an array that would have been defined as
   $a = array( 1=>’one’, 3=>’three’);
   and NOT
   $a = array( 1 => ’one’, 2 => ’three’);
*/
```

The foreach control structure exists specifically for arrays. It provides an easy way to traverse an array.

Array do’s and don’ts

Why is `$foo[bar]` wrong?

You should always use quotes around an associative array index. For example, use $foo[’bar’] and not $foo[bar]. But why is $foo[bar] wrong? You might have seen the following syntax in old scripts:

```
$foo[bar] = ’enemy’;
echo $foo[bar];
// etc
```

This is wrong, but it works. Then, why is it wrong? The reason is that this code has an undefined constant (bar) rather than a string (’bar’ - notice the quotes), and PHP may in future define constants which, unfortunately for your code, have the same name. It works, because the undefined constant gets converted to a string of the same name.

As stated in the syntax section, there must be an expression between the square brackets (’`[`’ and ’`]`’). That means that you can write things like this:

```
echo $arr[ foo(true) ];
```

This is an example of using a function return value as the array index. PHP knows also about constants, and you may have seen the `E_*` before.

```
$error_descriptions[E_ERROR] = "A fatal error has occured";
$error_descriptions[E_WARNING] = "PHP issued a warning";
$error_descriptions[E_NOTICE] = "This is just an informal notice";
```

Note that `E_ERROR` is also a valid identifier, just like `bar` in the first example. But the last example is in fact the same as writing:

```
$error_descriptions[1] = "A fatal error has occured";
```

```
$error_descriptions[2] = "PHP issued a warning";
$error_descriptions[8] = "This is just an informal notice";
```

because `E_ERROR` equals `1`, etc.

Then, how is it possible that `$foo[bar]` works? It works, because `bar` is due to its syntax expected to be a constant expression. However, in this case no constant with the name `bar` exists. PHP now assumes that you meant `bar` literally, as the string `"bar"`, but that you forgot to write the quotes.

So why is it bad then?

At some point in the future, the PHP team might want to add another constant or keyword, and then you get in trouble. For example, you already cannot use the words `empty` and `default` this way, since they are special reserved keywords.

> **Note:** When you turn error_reporting to `E_ALL`, you will see that PHP generates notices whenever an `index` is used which is not defined (put the line `error_reporting(E_ALL);` in your script).

> **Note:** Inside a double-quoted string, an other syntax is valid. See variable parsing in strings for more details.

Examples

The array type in PHP is very versatile, so here will be some examples to show you the full power of arrays.

```
// this
$a = array( ’color’ => ’red’
          , ’taste’ => ’sweet’
          , ’shape’ => ’round’
          , ’name’  => ’apple’
          ,            4        // key will be 0
          );

// is completely equivalent with
$a[’color’] = ’red’;
$a[’taste’] = ’sweet’;
$a[’shape’] = ’round’;
$a[’name’] = ’apple’;
$a[]        = 4;        // key will be 0

$b[] = ’a’;
```

```
$b[] = ’b’;
$b[] = ’c’;
// will result in the array array( 0 => ’a’ , 1 => ’b’ , 2 => ’c’ ),
// or simply array(’a’, ’b’, ’c’)
```

Example 7-4. Using array()

```
// Array as (property-)map
$map = array( ’version’    => 4
            , ’OS’         => ’Linux’
            , ’lang’       => ’english’
            , ’short_tags’ => true
            );

// strictly numerical keys
$array = array( 7
              , 8
              , 0
              , 156
              , -10
              );
// this is the same as array( 0 => 7, 1 => 8, ...)

$switching = array(         10 // key = 0
                  , 5    =>  6
                  , 3    =>  7
                  , ’a’  =>  4
                  ,         11 // key = 6 (maximum of integer-indices was 5)
                  , ’8’  =>  2 // key = 8 (integer!)
                  , ’02’ => 77 // key = ’02’
                  , 0    => 12 // the value 10 will be overwritten by 12
                  );

// empty array
$empty = array();
```

Example 7-5. Collection

```
$colors = array(’red’,’blue’,’green’,’yellow’);

foreach ( $colors as $color ) {
    echo "Do you like $color?\n";
}
```

```
/* output:
Do you like red?
Do you like blue?
Do you like green?
Do you like yellow?
*/
```

Note that it is currently not possible to change the values of the array directly in such a loop. A workaround is the following:

Example 7-6. Collection

```
foreach ($colors as $key => $color) {
    // won’t work:
    //$color = strtoupper($color);

    //works:
    $colors[$key] = strtoupper($color);
}
print_r($colors);

/* output:
Array
(
    [0] => RED
    [1] => BLUE
    [2] => GREEN
    [3] => YELLOW
)
*/
```

This example creates a one-based array.

Example 7-7. One-based index

```
$firstquarter  = array(1 => ’January’, ’February’, ’March’);
print_r($firstquarter);

/* output:
Array
(
    [1] => ’January’
    [2] => ’February’
    [3] => ’March’
)
*/
```

Example 7-8. Filling real array

```
// fill an array with all items from a directory
$handle = opendir(’.’);
while ($file = readdir($handle))
{
    $files[] = $file;
}
closedir($handle);
```

Arrays are ordered. You can also change the order using various sorting-functions. See array-functions for more information.

Example 7-9. Sorting array

```
sort($files);
print_r($files);
```

Because the value of an array can be everything, it can also be another array. This way you can make recursive and multi-dimensional arrays.

Example 7-10. Recursive and multi-dimensional arrays

```
$fruits = array ( "fruits"  => array ( "a" => "orange"
                                     , "b" => "banana"
                                     , "c" => "apple"
                                     )
                , "numbers" => array ( 1
                                     , 2
                                     , 3
                                     , 4
                                     , 5
                                     , 6
                                     )
                , "holes"   => array (      "first"
                                     , 5 => "second"
                                     ,      "third"
                                     )
                  );
```

Objects

Object Initialization

To initialize an object, you use the `new` statement to instantiate the object to a variable.

```
<?php
class foo
{
    function do_foo()
    {
        echo "Doing foo.";
    }
}

$bar = new foo;
$bar->do_foo();
?>
```

For a full discussion, please read the section Classes and Objects.

Resource

A resource is a special variable, holding a reference to an external resource. Resources are created and used by special functions. See the appendix for a listing of all these functions and the corresponding resource types.

> **Note:** The resource type was introduced in PHP 4

Freeing resources

Due to the reference-counting system introduced with PHP4's Zend-engine, it is automatically detected when a resource is no longer referred to (just like Java). When this is the case, all resources that were in use for this resource are made free by the garbage collector. For this reason, it is rarely ever necessary to free the memory manually by using some free_result function.

> **Note:** Persistent database-links are special, they are *not* destroyed by the gc. See also persistent links

NULL

The special `NULL` value represents that a variable has no value. `NULL` is the only possible value of type NULL.

> **Note:** The null type was introduced in PHP 4

A variable is considered to be `NULL` if

- it has been assigned the constant `NULL`.
- it has not been set to any value yet.
- it has been unset().

Syntax

There is only one value of type `NULL`, and that is the case-insensitive keyword `NULL`.

```
$var = NULL;
```

See also is_null() and unset().

Type Juggling

PHP does not require (or support) explicit type definition in variable declaration; a variable's type is determined by the context in which that variable is used. That is to say, if you assign a string value to variable `var`, `var` becomes a string. If you then assign an integer value to `var`, it becomes an integer.

An example of PHP's automatic type conversion is the addition operator '+'. If any of the operands is a float, then all operands are evaluated as floats, and the result will be a float. Otherwise, the operands will be interpreted as integers, and the result will also be an integer. Note that this does NOT change the types of the operands themselves; the only change is in how the operands are evaluated.

```
$foo = "0";  // $foo is string (ASCII 48)

$foo += 2;   // $foo is now an integer (2)
$foo = $foo + 1.3;  // $foo is now a float (3.3)
$foo = 5 + "10 Little Piggies"; // $foo is integer (15)
$foo = 5 + "10 Small Pigs";     // $foo is integer (15)
```

If the last two examples above seem odd, see String conversion.

If you wish to force a variable to be evaluated as a certain type, see the section on Type casting. If you wish to change the type of a variable, see settype().

If you would like to test any of the examples in this section, you can use the var_dump() function.

Note: The behaviour of an automatic conversion to array is currently undefined.

```
$a = "1";     // $a is a string
$a[0] = "f";  // What about string offsets? What happens?
```

Since PHP supports indexing into strings via offsets using the same syntax as array indexing, the example above leads to a problem: should $a become an array with its first element being "f", or should "f" become the first character of the string $a?

For this reason, as of PHP 3.0.12 and PHP 4.0b3-RC4, the result of this automatic conversion is considered to be undefined. Fixes are, however, being discussed.

Type Casting

Type casting in PHP works much as it does in C: the name of the desired type is written in parentheses before the variable which is to be cast.

```
$foo = 10;   // $foo is an integer
$bar = (float) $foo;   // $bar is a float
```

The casts allowed are:

- (int), (integer) - cast to integer
- (bool), (boolean) - cast to boolean
- (float), (double), (real) - cast to float
- (string) - cast to string
- (array) - cast to array
- (object) - cast to object

Note: Instead of casting a variable to string, you can also enclose the variable in double quotes.

Note that tabs and spaces are allowed inside the parentheses, so the following are functionally equivalent:

```
$foo = (int) $bar;
$foo = ( int ) $bar;
```

It may not be obvious exactly what will happen when casting between certain types. For more info, see these sections:

- Converting to boolean
- Converting to integer

When casting or forcing a conversion from array to string, the result will be the word `Array`. When casting or forcing a conversion from object to string, the result will be the word `Object`.

When casting from a scalar or a string variable to an array, the variable will become the first element of the array:

```
$var = ’ciao’;
$arr = (array) $var;
echo $arr[0];  // outputs ’ciao’
```

When casting from a scalar or a string variable to an object, the variable will become an attribute of the object; the attribute name will be ’scalar’:

```
$var = ’ciao’;
```

```
$obj = (object) $var;
echo $obj->scalar;  // outputs ’ciao’
```

Chapter 8. Variables

Basics

Variables in PHP are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.

Variable names follow the same rules as other labels in PHP. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: ’[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*’

> **Note:** For our purposes here, a letter is a-z, A-Z, and the ASCII characters from 127 through 255 (0x7f-0xff).

```
<?php
$var = "Bob";
$Var = "Joe";
echo "$var, $Var";      // outputs "Bob, Joe"

$4site = ’not yet’;     // invalid; starts with a number
$_4site = ’not yet’;    // valid; starts with an underscore
$täyte = ’mansikka’;    // valid; ’ä’ is ASCII 228.
?>
```

In PHP 3, variables are always assigned by value. That is to say, when you assign an expression to a variable, the entire value of the original expression is copied into the destination variable. This means, for instance, that after assigning one variable’s value to another, changing one of those variables will have no effect on the other. For more information on this kind of assignment, see the chapter on Expressions.

PHP 4 offers another way to assign values to variables: assign by reference. This means that the new variable simply references (in other words, "becomes an alias for" or "points to") the original variable. Changes to the new variable affect the original, and vice versa. This also means that no copying is performed; thus, the assignment happens more quickly. However, any speedup will likely be noticed only in tight loops or when assigning large arrays or objects.

To assign by reference, simply prepend an ampersand (&) to the beginning of the variable which is being assigned (the source variable). For instance, the following code snippet outputs ’My name is Bob’ twice:

```
<?php
$foo = ’Bob’;              // Assign the value ’Bob’ to $foo
$bar = &$foo;              // Reference $foo via $bar.
$bar = "My name is $bar";  // Alter $bar...
echo $bar;
echo $foo;                 // $foo is altered too.
?>
```

One important thing to note is that only named variables may be assigned by reference.

```
<?php
$foo = 25;
$bar = &$foo;      // This is a valid assignment.
$bar = &(24 * 7);  // Invalid; references an unnamed expression.

function test()
{
   return 25;
}

$bar = &test();    // Invalid.
?>
```

Predefined variables

PHP provides a large number of predefined variables to any script which it runs. Many of these variables, however, cannot be fully documented as they are dependent upon which server is running, the version and setup of the server, and other factors. Some of these variables will not be available when PHP is run on the command line. For a listing of these variables, please see the section on Reserved Predefined Variables.

Warning

In PHP 4.2.0 and later, the default value for the PHP directive register_globals is *off*. This is a major change in PHP. Having register_globals *off* affects the set of predefined variables available in the global scope. For example, to get `DOCUMENT_ROOT` you'll use `$_SERVER['DOCUMENT_ROOT']` instead of `$DOCUMENT_ROOT`, or `$_GET['id']` from the URL `http://www.example.com/test.php?id=3` instead of `$id`, or `$_ENV['HOME']` instead of `$HOME`.

For related information on this change, read the configuration entry for register_globals, the security chapter on Using Register Globals , as well as the PHP 4.1.0 (http://www.php.net/release_4_1_0.php) and 4.2.0 (http://www.php.net/release_4_2_0.php) Release Announcements.

Using the available PHP Reserved Predefined Variables, like the superglobal arrays, is preferred.

From version 4.1.0 onward, PHP provides an additional set of predefined arrays containing variables from the web server (if applicable), the environment, and user input. These new arrays are rather special in that they are automatically global--i.e., automatically available in every scope. For this reason, they are often known as ’autoglobals’ or ’superglobals’. (There is no mechanism in PHP for user-defined superglobals.) The superglobals are listed below; however, for a listing of their contents and further discussion on PHP predefined variables and their natures, please see the section Reserved Predefined Variables. Also, you’ll notice how the older predefined variables (`$HTTP_*_VARS`) still exist.

If certain variables in variables_order are not set, their appropriate PHP predefined arrays are also left empty.

PHP Superglobals

$GLOBALS

Contains a reference to every variable which is currently available within the global scope of the script. The keys of this array are the names of the global variables. `$GLOBALS` has existed since PHP 3.

$_SERVER

Variables set by the web server or otherwise directly related to the execution environment of the current script. Analogous to the old `$HTTP_SERVER_VARS` array (which is still available, but deprecated).

$_GET

Variables provided to the script via HTTP GET. Analogous to the old `$HTTP_GET_VARS` array (which is still available, but deprecated).

$_POST

Variables provided to the script via HTTP POST. Analogous to the old `$HTTP_POST_VARS` array (which is still available, but deprecated).

$_COOKIE

Variables provided to the script via HTTP cookies. Analogous to the old `$HTTP_COOKIE_VARS` array (which is still available, but deprecated).

$_FILES

Variables provided to the script via HTTP post file uploads. Analogous to the old `$HTTP_POST_FILES` array (which is still available, but deprecated). See POST method uploads for more information.

$_ENV

Variables provided to the script via the environment. Analogous to the old `$HTTP_ENV_VARS` array (which is still available, but deprecated).

$_REQUEST

Variables provided to the script via any user input mechanism, and which therefore cannot be trusted. The presence and order of variable inclusion in this array is defined according to the variables_order configuration directive. This array has no direct analogue in versions of PHP prior to 4.1.0. See also import_request_variables().

Note: When running on the command line , this will *not* include the `argv` and `argc` entries; these are present in the `$_SERVER` array.

$_SESSION

Variables which are currently registered to a script's session. Analogous to the old `$HTTP_SESSION_VARS` array (which is still available, but deprecated). See the Session handling functions section for more information.

Variable scope

The scope of a variable is the context within which it is defined. For the most part all PHP variables only have a single scope. This single scope spans included and required files as well. For example:

```
<?php
$a = 1;
include "b.inc";
?>
```

Here the `$a` variable will be available within the included `b.inc` script. However, within user-defined functions a local function scope is introduced. Any variable used inside a function is by default limited to the local function scope. For example:

```
<?php
$a = 1; /* global scope */

function Test()
{
    echo $a; /* reference to local scope variable */
}

Test();
?>
```

This script will not produce any output because the echo statement refers to a local version of the `$a` variable, and it has not been assigned a value within this scope. You may notice that this is a little bit different from the C language in that global variables in C are automatically available to functions unless specifically overridden by a local definition. This can cause some problems in that people may inadvertently change a global variable. In PHP global variables must be declared global inside a function if they are going to be used in that function. An example:

```
<?php
$a = 1;
$b = 2;

function Sum()
{
    global $a, $b;

    $b = $a + $b;
}

Sum();
echo $b;
?>
```

The above script will output "3". By declaring `$a` and `$b` global within the function, all references to either variable will refer to the global version. There is no limit to the number of global variables that can be manipulated by a function.

A second way to access variables from the global scope is to use the special PHP-defined `$GLOBALS` array. The previous example can be rewritten as:

```
<?php
$a = 1;
$b = 2;

function Sum()
{
    $GLOBALS["b"] = $GLOBALS["a"] + $GLOBALS["b"];
}

Sum();
echo $b;
?>
```

The `$GLOBALS` array is an associative array with the name of the global variable being the key and the contents of that variable being the value of the array element. Notice how `$GLOBALS` exists in any scope, this is because $GLOBALS is a superglobal. Here's an example demonstrating the power of superglobals:

```
<?php
function test_global()
{
    // Most predefined variables aren't "super" and require
    // 'global' to be available to the functions local scope.
    global $HTTP_POST_VARS;

    print $HTTP_POST_VARS['name'];

    // Superglobals are available in any scope and do
    // not require 'global'.  Superglobals are available
    // as of PHP 4.1.0
    print $_POST['name'];
}
?>
```

Another important feature of variable scoping is the *static* variable. A static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope. Consider the following example:

```
<?php
function Test ()
{
    $a = 0;
    echo $a;
    $a++;
}
?>
```

This function is quite useless since every time it is called it sets `$a` to `0` and prints "0". The `$a++` which increments the variable serves no purpose since as soon as the function exits the `$a` variable disappears. To make a useful counting function which will not lose track of the current count, the `$a` variable is declared static:

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

Now, every time the Test() function is called it will print the value of `$a` and increment it.

Static variables also provide one way to deal with recursive functions. A recursive function is one which calls itself. Care must be taken when writing a recursive function because it is possible to make it recurse indefinitely. You must make sure you have an adequate way of terminating the recursion. The following simple function recursively counts to 10, using the static variable `$count` to know when to stop:

```
<?php
function Test()
{
    static $count = 0;

    $count++;
    echo $count;
    if ($count < 10) {
        Test ();
    }
    $count--;
}
?>
```

The Zend Engine 1, driving `PHP4`, implements the `static` and `global` modifier for variables in terms of references. For example, a true global variable imported inside a function scope with the `global` statement actually creates a reference to the global variable. This can lead to unexpected behaviour which the following example addresses:

```
<?php
function test_global_ref() {
    global $obj;
    $obj = &new stdclass;
}

function test_global_noref() {
    global $obj;
    $obj = new stdclass;
}

test_global_ref();
var_dump($obj);
test_global_noref();
var_dump($obj);
?>
```

Executing this example will result in the following output:

```
NULL
object(stdClass)(0) {
}
```

A similar behaviour applies to the `static` statement. References are not stored statically:

```
<?php
function &get_instance_ref() {
    static $obj;

    echo "Static object: ";
    var_dump($obj);
    if (!isset($obj)) {
        // Assign a reference to the static variable
        $obj = &new stdclass;
    }
    $obj->property++;
    return $obj;
}

function &get_instance_noref() {
    static $obj;

    echo "Static object: ";
    var_dump($obj);
    if (!isset($obj)) {
        // Assign the object to the static variable
        $obj = new stdclass;
    }
    $obj->property++;
    return $obj;
}

$obj1 = get_instance_ref();
$still_obj1 = get_instance_ref();
echo "\n";
$obj2 = get_instance_noref();
$still_obj2 = get_instance_noref();
?>
```

Executing this example will result in the following output:

```
Static object: NULL
Static object: NULL

Static object: NULL
Static object: object(stdClass)(1) {
  ["property"]=>
  int(1)
}
```

This example demonstrates that when assigning a reference to a static variable, it's not *remembered* when you call the `&get_instance_ref()` function a second time.

Variable variables

Sometimes it is convenient to be able to have variable variable names. That is, a variable name which can be set and used dynamically. A normal variable is set with a statement such as:

```
<?php
$a = "hello";
?>
```

A variable variable takes the value of a variable and treats that as the name of a variable. In the above example, *hello*, can be used as the name of a variable by using two dollar signs. i.e.

```
<?php
$$a = "world";
?>
```

At this point two variables have been defined and stored in the PHP symbol tree: `$a` with contents "hello" and `$hello` with contents "world". Therefore, this statement:

```
<?php
echo "$a ${$a}";
?>
```

produces the exact same output as:

```
<?php
echo "$a $hello";
?>
```

i.e. they both produce: `hello world`.

In order to use variable variables with arrays, you have to resolve an ambiguity problem. That is, if you write `$$a[1]` then the parser needs to know if you meant to use `$a[1]` as a variable, or if you wanted `$$a` as the variable and then the [1] index from that variable. The syntax for resolving this ambiguity is: `${$a[1]}` for the first case and `${$a}[1]` for the second.

Warning

Please note that variable variables cannot be used with PHP's Superglobal arrays. This means you cannot do things like `${$_GET}`. If you are looking for a way to handle availability of superglobals and the old `HTTP_*_VARS`, you might want to try referencing them.

Variables from outside PHP

HTML Forms (GET and POST)

When a form is submitted to a PHP script, the information from that form is automatically made available to the script. There are many ways to access this information, for example:

Example 8-1. A simple HTML form

```
<form action="foo.php" method="post">
    Name:  <input type="text" name="username"><br>
    Email: <input type="text" name="email"><br>
    <input type="submit" name="submit" value="Submit me!">
</form>
```

Depending on your particular setup and personal preferences, there are many ways to access data from your HTML forms. Some examples are:

Example 8-2. Accessing data from a simple POST HTML form

```
<?php
// Available since PHP 4.1.0

   print $_POST['username'];
   print $_REQUEST['username'];

   import_request_variables('p', 'p_');
   print $p_username;

// Available since PHP 3.

   print $HTTP_POST_VARS['username'];

// Available if the PHP directive register_globals = on.  As of
// PHP 4.2.0 the default value of register_globals = off.
// Using/relying on this method is not preferred.

   print $username;
?>
```

Using a GET form is similar except you’ll use the appropriate GET predefined variable instead. GET also applies to the QUERY_STRING (the information after the ’?’ in an URL). So, for example, `http://www.example.com/test.php?id=3` contains GET data which is accessible with `$_GET[’id’]`. See also $_REQUEST and import_request_variables().

> **Note:** Superglobal arrays, like `$_POST` and `$_GET`, became available in PHP 4.1.0

As shown, before PHP 4.2.0 the default value for register_globals was *on*. And, in PHP 3 it was always on. The PHP community is encouraging all to not rely on this directive as it’s preferred to assume it’s *off* and code accordingly.

> **Note:** The magic_quotes_gpc configuration directive affects Get, Post and Cookie values. If turned on, value (It’s "PHP!") will automagically become (It\’s \"PHP!\"). Escaping is needed for DB insertion. See also addslashes(), stripslashes() and magic_quotes_sybase.

PHP also understands arrays in the context of form variables (see the related faq). You may, for example, group related variables together, or use this feature to retrieve values from a multiple select input. For example, let’s post a form to itself and upon submission display the data:

Example 8-3. More complex form variables

```
<?php
if ($HTTP_POST_VARS[’action’] == ’submitted’) {
    print ’<pre>’;

    print_r($HTTP_POST_VARS);
    print ’<a href="’. $HTTP_SERVER_VARS[’PHP_SELF’] .’">Please try again</a>’;

    print ’</pre>’;
} else {
?>
<form action="<?php echo $HTTP_SERVER_VARS[’PHP_SELF’]; ?>" method="post">
    Name:  <input type="text" name="personal[name]"><br>
    Email: <input type="text" name="personal[email]"><br>
    Beer: <br>
    <select multiple name="beer[]">
        <option value="warthog">Warthog</option>
        <option value="guinness">Guinness</option>
        <option value="stuttgarter">Stuttgarter Schwabenbräu</option>
    </select><br>
    <input type="hidden" name="action" value="submitted">
    <input type="submit" name="submit" value="submit me!">
</form>
<?php
}
?>
```

In PHP 3, the array form variable usage is limited to single-dimensional arrays. In PHP 4, no such restriction applies.

IMAGE SUBMIT variable names

When submitting a form, it is possible to use an image instead of the standard submit button with a tag like:

```
<input type="image" src="image.gif" name="sub">
```

When the user clicks somewhere on the image, the accompanying form will be transmitted to the server with two additional variables, sub_x and sub_y. These contain the coordinates of the user click within the image. The experienced may note that the actual variable names sent by the browser contains a period rather than an underscore, but PHP converts the period to an underscore automatically.

HTTP Cookies

PHP transparently supports HTTP cookies as defined by Netscape's Spec (http://www.netscape.com/newsref/std/cookie_spec.html). Cookies are a mechanism for storing data in the remote browser and thus tracking or identifying return users. You can set cookies using the setcookie() function. Cookies are part of the HTTP header, so the SetCookie function must be called before any output is sent to the browser. This is the same restriction as for the header() function. Cookie data is then available in the appropriate cookie data arrays, such as `$_COOKIE`, `$HTTP_COOKIE_VARS` as well as in `$_REQUEST`. See the setcookie() manual page for more details and examples.

If you wish to assign multiple values to a single cookie variable, you may assign it as an array. For example:

```
<?php
  setcookie("MyCookie[foo]", "Testing 1", time()+3600);
  setcookie("MyCookie[bar]", "Testing 2", time()+3600);
?>
```

That will create two seperate cookies although MyCookie will now be a single array in your script. If you want to set just one cookie with multiple values, consider using serialize() or explode() on the value first.

Note that a cookie will replace a previous cookie by the same name in your browser unless the path or domain is different. So, for a shopping cart application you may want to keep a counter and pass this along. i.e.

Example 8-4. A setcookie() example

```
<?php
$count++;
setcookie("count", $count, time()+3600);
setcookie("Cart[$count]", $item, time()+3600);
?>
```

Dots in incoming variable names

Typically, PHP does not alter the names of variables when they are passed into a script. However, it should be noted that the dot (period, full stop) is not a valid character in a PHP variable name. For the reason, look at it:

```
<?php
$varname.ext;  /* invalid variable name */
?>
```

Now, what the parser sees is a variable named `$varname`, followed by the string concatenation operator, followed by the barestring (i.e. unquoted string which doesn’t match any known key or reserved words) ’ext’. Obviously, this doesn’t have the intended result.

For this reason, it is important to note that PHP will automatically replace any dots in incoming variable names with underscores.

Determining variable types

Because PHP determines the types of variables and converts them (generally) as needed, it is not always obvious what type a given variable is at any one time. PHP includes several functions which find out what type a variable is, such as: gettype(), is_array(), is_float(), is_int(), is_object(), and is_string(). See also the chapter on Types.

Chapter 9. Constants

A constant is a identifier (name) for a simple value. As the name suggests, that value cannot change during the execution of the script. (The ’magic constants’ `__FILE__` and `__LINE__` appear to be an exception to this, but they’re not actually constants.) A constant is case-sensitive by default. By convention constant identifiers are always uppercase.

The name of a constant follows the same rules as any label in PHP. A valid constant name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: `[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*`

> **Note:** For our purposes here, a letter is a-z, A-Z, and the ASCII characters from 127 through 255 (0x7f-0xff).

The scope of a constant is global--you can access it anywhere in your script without regard to scope.

Syntax

You can define a constant by using the define()-function. Once a constant is defined, it can never be changed or undefined.

Only scalar data (boolean, integer, float and string) can be contained in constants.

You can get the value of a constant by simply specifying its name. Unlike with variables, you should *not* prepend a constant with a `$`. You can also use the function constant(), to read a constant’s value, if you are to obtain the constant’s name dynamically. Use get_defined_constants() to get a list of all defined constants.

> **Note:** Constants and (global) variables are in a different namespace. This implies that for example `TRUE` and `$TRUE` are generally different.

If you use an undefined constant, PHP assumes that you mean the name of the constant itself. A notice will be issued when this happens. Use the defined()-function if you want to know if a constant is set.

These are the differences between constants and variables:

- Constants do not have a dollar sign (`$`) before them;
- Constants may only be defined using the define() function, not by simple assignment;
- Constants may be defined and accessed anywhere without regard to variable scoping rules;
- Constants may not be redefined or undefined once they have been set; and
- Constants may only evaluate to scalar values.

Example 9-1. Defining Constants

```
<?php
```

```
define("CONSTANT", "Hello world.");
echo CONSTANT; // outputs "Hello world."
echo Constant; // outputs "Constant" and issues a notice.
?>
```

Predefined constants

PHP provides a large number of predefined constants to any script which it runs. Many of these constants, however, are created by various extensions, and will only be present when those extensions are available, either via dynamic loading or because they have been compiled in.

A list of predefined constants is available in the section Predefined constants.

Chapter 10. Expressions

Expressions are the most important building stones of PHP. In PHP, almost anything you write is an expression. The simplest yet most accurate way to define an expression is "anything that has a value".

The most basic forms of expressions are constants and variables. When you type "$a = 5", you’re assigning ’5’ into $a. ’5’, obviously, has the value 5, or in other words ’5’ is an expression with the value of 5 (in this case, ’5’ is an integer constant).

After this assignment, you’d expect $a’s value to be 5 as well, so if you wrote $b = $a, you’d expect it to behave just as if you wrote $b = 5. In other words, $a is an expression with the value of 5 as well. If everything works right, this is exactly what will happen.

Slightly more complex examples for expressions are functions. For instance, consider the following function:

```
function foo ()
{
    return 5;
}
```

Assuming you’re familiar with the concept of functions (if you’re not, take a look at the chapter about functions), you’d assume that typing `$c = foo()` is essentially just like writing `$c = 5`, and you’re right. Functions are expressions with the value of their return value. Since foo() returns 5, the value of the expression ’foo()’ is 5. Usually functions don’t just return a static value but compute something.

Of course, values in PHP don’t have to be integers, and very often they aren’t. PHP supports three scalar value types: integer values, floating point values and string values (scalar values are values that you can’t ’break’ into smaller pieces, unlike arrays, for instance). PHP also supports two composite (non-scalar) types: arrays and objects. Each of these value types can be assigned into variables or returned from functions.

So far, users of PHP/FI 2 shouldn’t feel any change. However, PHP takes expressions much further, in the same way many other languages do. PHP is an expression-oriented language, in the sense that almost everything is an expression. Consider the example we’ve already dealt with, ’$a = 5’. It’s easy to see that there are two values involved here, the value of the integer constant ’5’, and the value of $a which is being updated to 5 as well. But the truth is that there’s one additional value involved here, and that’s the value of the assignment itself. The assignment itself evaluates to the assigned value, in this case 5. In practice, it means that ’$a = 5’, regardless of what it does, is an expression with the value 5. Thus, writing something like ’$b = ($a = 5)’ is like writing ’$a = 5; $b = 5;’ (a semicolon marks the end of a statement). Since assignments are parsed in a right to left order, you can also write ’$b = $a = 5’.

Another good example of expression orientation is pre- and post-increment and decrement. Users of PHP/FI 2 and many other languages may be familiar with the notation of variable++ and variable--. These are increment and decrement operators. In PHP/FI 2, the statement ’$a++’ has no value (is not an expression), and thus you can’t assign it or use it in any way. PHP enhances the increment/decrement capabilities by making these expressions as well, like in C. In PHP, like in C, there are two types of increment - pre-increment and post-increment. Both pre-increment and post-increment essentially increment the variable, and the effect on the variable is idential. The difference is with the value of the increment expression. Pre-increment, which is written ’++$variable’, evaluates to the incremented value

(PHP increments the variable before reading its value, thus the name ’pre-increment’). Post-increment, which is written ’$variable++’ evaluates to the original value of $variable, before it was incremented (PHP increments the variable after reading its value, thus the name ’post-increment’).

A very common type of expressions are comparison expressions. These expressions evaluate to either 0 or 1, meaning `FALSE` or `TRUE` (respectively). PHP supports > (bigger than), >= (bigger than or equal to), == (equal), != (not equal), < (smaller than) and <= (smaller than or equal to). These expressions are most commonly used inside conditional execution, such as `if` statements.

The last example of expressions we’ll deal with here is combined operator-assignment expressions. You already know that if you want to increment $a by 1, you can simply write ’$a++’ or ’++$a’. But what if you want to add more than one to it, for instance 3? You could write ’$a++’ multiple times, but this is obviously not a very efficient or comfortable way. A much more common practice is to write ’$a = $a + 3’. ’$a + 3’ evaluates to the value of $a plus 3, and is assigned back into $a, which results in incrementing $a by 3. In PHP, as in several other languages like C, you can write this in a shorter way, which with time would become clearer and quicker to understand as well. Adding 3 to the current value of $a can be written ’$a += 3’. This means exactly "take the value of $a, add 3 to it, and assign it back into $a". In addition to being shorter and clearer, this also results in faster execution. The value of ’$a += 3’, like the value of a regular assignment, is the assigned value. Notice that it is NOT 3, but the combined value of $a plus 3 (this is the value that’s assigned into $a). Any two-place operator can be used in this operator-assignment mode, for example ’$a -= 5’ (subtract 5 from the value of $a), ’$b *= 7’ (multiply the value of $b by 7), etc.

There is one more expression that may seem odd if you haven’t seen it in other languages, the ternary conditional operator:

```
$first ? $second : $third
```

If the value of the first subexpression is `TRUE` (non-zero), then the second subexpression is evaluated, and that is the result of the conditional expression. Otherwise, the third subexpression is evaluated, and that is the value.

The following example should help you understand pre- and post-increment and expressions in general a bit better:

```
function double($i)
{
    return $i*2;
}
$b = $a = 5;        /* assign the value five into the variable $a and $b */
$c = $a++;          /* post-increment, assign original value of $a
                       (5) to $c */
$e = $d = ++$b;     /* pre-increment, assign the incremented value of
                       $b (6) to $d and $e */

/* at this point, both $d and $e are equal to 6 */

$f = double($d++);  /* assign twice the value of $d <emphasis>before</emphasis>
```

```
                           the increment, 2*6 = 12 to $f */
$g = double(++$e);  /* assign twice the value of $e <emphasis>after</emphasis>
                           the increment, 2*7 = 14 to $g */
$h = $g += 10;      /* first, $g is incremented by 10 and ends with the
                           value of 24. the value of the assignment (24) is
                           then assigned into $h, and $h ends with the value
                           of 24 as well. */
```

In the beginning of the chapter we said that we'll be describing the various statement types, and as promised, expressions can be statements. However, not every expression is a statement. In this case, a statement has the form of 'expr' ';' that is, an expression followed by a semicolon. In '$b=$a=5;', $a=5 is a valid expression, but it's not a statement by itself. '$b=$a=5;' however is a valid statement.

One last thing worth mentioning is the truth value of expressions. In many events, mainly in conditional execution and loops, you're not interested in the specific value of the expression, but only care about whether it means `TRUE` or `FALSE`. The constants `TRUE` and `FALSE` (case-insensitive) are the two possible boolean values. When necessary, an expression is automatically converted to boolean. See the section about type-casting for details about how.

PHP provides a full and powerful implementation of expressions, and documenting it entirely goes beyond the scope of this manual. The above examples should give you a good idea about what expressions are and how you can construct useful expressions. Throughout the rest of this manual we'll write *`expr`* to indicate any valid PHP expression.

Chapter 11. Operators

Operator Precedence

The precedence of an operator specifies how "tightly" it binds two expressions together. For example, in the expression `1 + 5 * 3`, the answer is `16` and not `18` because the multiplication ("*") operator has a higher precedence than the addition ("+") operator. Parentheses may be used to force precedence, if necessary. For instance: `(1 + 5) * 3` evaluates to `18`.

The following table lists the precedence of operators with the lowest-precedence operators listed first.

Table 11-1. Operator Precedence

Associativity	Operators
left	,
left	or
left	xor
left	and
right	print
left	= += -= *= /= .= %= &= |= ^= ~= <<= >>=
left	? :
left	||
left	&&
left	|
left	^
left	&
non-associative	== != === !==
non-associative	< <= > >=
left	<< >>
left	+ - .
left	* / %
right	! ~ ++ -- (int) (float) (string) (array) (object) @
right	[
non-associative	new

Arithmetic Operators

Remember basic arithmetic from school? These work just like those.

Table 11-2. Arithmetic Operators

Example	Name	Result
$a + $b	Addition	Sum of $a and $b.

Example	**Name**	**Result**
$a - $b	Subtraction	Difference of $a and $b.
$a * $b	Multiplication	Product of $a and $b.
$a / $b	Division	Quotient of $a and $b.
$a % $b	Modulus	Remainder of $a divided by $b.

The division operator ("/") returns a float value anytime, even if the two operands are integers (or strings that get converted to integers).

Assignment Operators

The basic assignment operator is "=". Your first inclination might be to think of this as "equal to". Don't. It really means that the the left operand gets set to the value of the expression on the rights (that is, "gets set to").

The value of an assignment expression is the value assigned. That is, the value of "$a = 3" is 3. This allows you to do some tricky things:

```
$a = ($b = 4) + 5; // $a is equal to 9 now, and $b has been set to 4.
```

In addition to the basic assignment operator, there are "combined operators" for all of the binary arithmetic and string operators that allow you to use a value in an expression and then set its value to the result of that expression. For example:

```
$a = 3;
$a += 5; // sets $a to 8, as if we had said: $a = $a + 5;
$b = "Hello ";
$b .= "There!"; // sets $b to "Hello There!", just like $b = $b . "There!";
```

Note that the assignment copies the original variable to the new one (assignment by value), so changes to one will not affect the other. This may also have relevance if you need to copy something like a large array inside a tight loop. PHP 4 supports assignment by reference, using the `$var = &$othervar;` syntax, but this is not possible in PHP 3. 'Assignment by reference' means that both variables end up pointing at the same data, and nothing is copied anywhere. To learn more about references, please read References explained.

Bitwise Operators

Bitwise operators allow you to turn specific bits within an integer on or off. If both the left- and right-hand parameters are strings, the bitwise operator will operate on the characters in this string.

```
<?php
    echo 12 ^ 9; // Outputs ’5’

    echo "12" ^ "9"; // Outputs the Backspace character (ascii 8)
                     // (’1’ (ascii 49)) ^ (’9’ (ascii 57)) = #8

    echo "hallo" ^ "hello"; // Outputs the ascii values #0 #4 #0 #0 #0
                            // ’a’ ^ ’e’ = #4
?>
```

Table 11-3. Bitwise Operators

Example	Name	Result
$a & $b	And	Bits that are set in both $a and $b are set.
$a | $b	Or	Bits that are set in either $a or $b are set.
$a ^ $b	Xor	Bits that are set in $a or $b but not both are set.
~ $a	Not	Bits that are set in $a are not set, and vice versa.
$a << $b	Shift left	Shift the bits of $a $b steps to the left (each step means "multiply by two")
$a >> $b	Shift right	Shift the bits of $a $b steps to the right (each step means "divide by two")

Comparison Operators

Comparison operators, as their name implies, allow you to compare two values.

Table 11-4. Comparison Operators

Example	Name	Result
$a == $b	Equal	`TRUE` if $a is equal to $b.

Example	Name	Result
$a === $b	Identical	`TRUE` if $a is equal to $b, and they are of the same type. (PHP 4 only)
$a != $b	Not equal	`TRUE` if $a is not equal to $b.
$a <> $b	Not equal	`TRUE` if $a is not equal to $b.
$a !== $b	Not identical	`TRUE` if $a is not equal to $b, or they are not of the same type. (PHP 4 only)
$a < $b	Less than	`TRUE` if $a is strictly less than $b.
$a > $b	Greater than	`TRUE` if $a is strictly greater than $b.
$a <= $b	Less than or equal to	`TRUE` if $a is less than or equal to $b.
$a >= $b	Greater than or equal to	`TRUE` if $a is greater than or equal to $b.

Another conditional operator is the "?:" (or ternary) operator, which operates as in C and many other languages.

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

This expression evaluates to *`expr2`* if *`expr1`* evaluates to `TRUE`, and *`expr3`* if *`expr1`* evaluates to `FALSE`.

Error Control Operators

PHP supports one error control operator: the at sign (@). When prepended to an expression in PHP, any error messages that might be generated by that expression will be ignored.

If the track_errors feature is enabled, any error message generated by the expression will be saved in the global variable $php_errormsg. This variable will be overwritten on each error, so check early if you want to use it.

```
<?php
/* Intentional file error */
$my_file = @file ('non_existent_file') or
    die ("Failed opening file: error was '$php_errormsg'");

// this works for any expression, not just functions:
$value = @$cache[$key];
// will not issue a notice if the index $key doesn't exist.
```

```
?>
```

Note: The @-operator works only on expressions. A simple rule of thumb is: if you can take the value of something, you can prepend the @ operator to it. For instance, you can prepend it to variables, function and include() calls, constants, and so forth. You cannot prepend it to function or class definitions, or conditional structures such as `if` and `foreach`, and so forth.

See also error_reporting().

Warning

Currently the "@" error-control operator prefix will even disable error reporting for critical errors that will terminate script execution. Among other things, this means that if you use "@" to suppress errors from a certain function and either it isn't available or has been mistyped, the script will die right there with no indication as to why.

Execution Operators

PHP supports one execution operator: backticks (“). Note that these are not single-quotes! PHP will attempt to execute the contents of the backticks as a shell command; the output will be returned (i.e., it won't simply be dumped to output; it can be assigned to a variable).

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

Note: The backtick operator is disabled when safe mode is enabled or shell_exec() is disabled.

See also escapeshellcmd(), exec(), passthru(), popen(), shell_exec(), and system().

Incrementing/Decrementing Operators

PHP supports C-style pre- and post-increment and decrement operators.

Table 11-5. Increment/decrement Operators

Example	Name	Effect
++$a	Pre-increment	Increments $a by one, then returns $a.
$a++	Post-increment	Returns $a, then increments $a by one.
--$a	Pre-decrement	Decrements $a by one, then returns $a.
$a--	Post-decrement	Returns $a, then decrements $a by one.

Here’s a simple example script:

```
<?php
echo "<h3>Postincrement</h3>";
$a = 5;
echo "Should be 5: " . $a++ . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";

echo "<h3>Preincrement</h3>";
$a = 5;
echo "Should be 6: " . ++$a . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";

echo "<h3>Postdecrement</h3>";
$a = 5;
echo "Should be 5: " . $a-- . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";

echo "<h3>Predecrement</h3>";
$a = 5;
echo "Should be 4: " . --$a . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";
?>
```

Logical Operators

Table 11-6. Logical Operators

Example	Name	Result
$a and $b	And	`TRUE` if both $a and $b are `TRUE`.

Example	**Name**	**Result**
$a or $b	Or	`TRUE` if either $a or $b is `TRUE`.
$a xor $b	Xor	`TRUE` if either $a or $b is `TRUE`, but not both.
! $a	Not	`TRUE` if $a is not `TRUE`.
$a && $b	And	`TRUE` if both $a and $b are `TRUE`.
$a || $b	Or	`TRUE` if either $a or $b is `TRUE`.

The reason for the two different variations of "and" and "or" operators is that they operate at different precedences. (See Operator Precedence.)

String Operators

There are two string operators. The first is the concatenation operator ('.'), which returns the concatenation of its right and left arguments. The second is the concatenating assignment operator ('.='), which appends the argument on the right side to the argument on the left side. Please read Assignment Operators for more information.

```
$a = "Hello ";
$b = $a . "World!"; // now $b contains "Hello World!"

$a = "Hello ";
$a .= "World!";     // now $a contains "Hello World!"
```

Array Operators

The only array operator in PHP is the + operator. It appends the right handed array to the left handed, whereas duplicated keys are NOT overwritten.

```
$a = array("a" => "apple", "b" => "banana");
$b = array("a" =>"pear", "b" => "strawberry", "c" => "cherry");

$c = $a + $b;

var_dump($c);
```

```
array(3) {
  ["a"]=>
  string(5) "apple"
  ["b"]=>
  string(6) "banana"
  ["c"]=>
  string(6) "cherry"
}
```

Chapter 12. Control Structures

Any PHP script is built out of a series of statements. A statement can be an assignment, a function call, a loop, a conditional statement of even a statement that does nothing (an empty statement). Statements usually end with a semicolon. In addition, statements can be grouped into a statement-group by encapsulating a group of statements with curly braces. A statement-group is a statement by itself as well. The various statement types are described in this chapter.

if

The `if` construct is one of the most important features of many languages, PHP included. It allows for conditional execution of code fragments. PHP features an `if` structure that is similar to that of C:

```
if (expr)
    statement
```

As described in the section about expressions, *`expr`* is evaluated to its Boolean value. If *`expr`* evaluates to `TRUE`, PHP will execute *`statement`*, and if it evaluates to `FALSE` - it’ll ignore it. More information about what values evaluate to `FALSE` can be found in the ’Converting to boolean’ section.

The following example would display `a is bigger than b` if *`$a`* is bigger than *`$b`*:

```
if ($a > $b)
    print "a is bigger than b";
```

Often you’d want to have more than one statement to be executed conditionally. Of course, there’s no need to wrap each statement with an `if` clause. Instead, you can group several statements into a statement group. For example, this code would display `a is bigger than b` if *`$a`* is bigger than *`$b`*, and would then assign the value of *`$a`* into *`$b`*:

```
if ($a > $b) {
    print "a is bigger than b";
    $b = $a;
}
```

`If` statements can be nested indefinitely within other `if` statements, which provides you with complete flexibility for conditional execution of the various parts of your program.

else

Often you'd want to execute a statement if a certain condition is met, and a different statement if the condition is not met. This is what `else` is for. `else` extends an `if` statement to execute a statement in case the expression in the `if` statement evaluates to `FALSE`. For example, the following code would display `a is bigger than b` if *$a* is bigger than *$b*, and `a is NOT bigger than b` otherwise:

```
if ($a > $b) {
    print "a is bigger than b";
} else {
    print "a is NOT bigger than b";
}
```

The `else` statement is only executed if the `if` expression evaluated to `FALSE`, and if there were any `elseif` expressions - only if they evaluated to `FALSE` as well (see elseif).

elseif

`elseif`, as its name suggests, is a combination of `if` and `else`. Like `else`, it extends an `if` statement to execute a different statement in case the original `if` expression evaluates to `FALSE`. However, unlike `else`, it will execute that alternative expression only if the `elseif` conditional expression evaluates to `TRUE`. For example, the following code would display `a is bigger than b`, `a equal to b` or `a is smaller than b`:

```
if ($a > $b) {
    print "a is bigger than b";
} elseif ($a == $b) {
    print "a is equal to b";
} else {
    print "a is smaller than b";
}
```

There may be several `elseif`s within the same `if` statement. The first `elseif` expression (if any) that evaluates to `TRUE` would be executed. In PHP, you can also write 'else if' (in two words) and the behavior would be identical to the one of 'elseif' (in a single word). The syntactic meaning is slightly different (if you're familiar with C, this is the same behavior) but the bottom line is that both would result in exactly the same behavior.

The `elseif` statement is only executed if the preceding `if` expression and any preceding `elseif` expressions evaluated to `FALSE`, and the current `elseif` expression evaluated to `TRUE`.

Alternative syntax for control structures

PHP offers an alternative syntax for some of its control structures; namely, `if`, `while`, `for`, `foreach`, and `switch`. In each case, the basic form of the alternate syntax is to change the opening brace to a colon (:) and the closing brace to `endif;`, `endwhile;`, `endfor;`, `endforeach;`, or `endswitch;`, respectively.

```
<?php if ($a == 5): ?>
A is equal to 5
<?php endif; ?>
```

In the above example, the HTML block "A is equal to 5" is nested within an `if` statement written in the alternative syntax. The HTML block would be displayed only if $a is equal to 5.

The alternative syntax applies to `else` and `elseif` as well. The following is an `if` structure with `elseif` and `else` in the alternative format:

```
if ($a == 5):
    print "a equals 5";
    print "...";
elseif ($a == 6):
    print "a equals 6";
    print "!!!";
else:
    print "a is neither 5 nor 6";
endif;
```

See also while, for, and if for further examples.

`while`

`while` loops are the simplest type of loop in PHP. They behave just like their C counterparts. The basic form of a `while` statement is:

```
while (expr) statement
```

The meaning of a `while` statement is simple. It tells PHP to execute the nested statement(s) repeatedly, as long as the `while` expression evaluates to `TRUE`. The value of the expression is checked each time at the beginning of the loop, so even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration (each time PHP runs the statements in the loop is one iteration). Sometimes, if the `while` expression evaluates to `FALSE` from the very beginning, the nested statement(s) won't even be run once.

Like with the `if` statement, you can group multiple statements within the same `while` loop by surrounding a group of statements with curly braces, or by using the alternate syntax:

```
while (expr): statement ... endwhile;
```

The following examples are identical, and both print numbers from 1 to 10:

```
/* example 1 */

$i = 1;
while ($i <= 10) {
    print $i++;  /* the printed value would be
                    $i before the increment
                    (post-increment) */
}

/* example 2 */

$i = 1;
while ($i <= 10):
    print $i;
    $i++;
endwhile;
```

do..while

`do..while` loops are very similar to `while` loops, except the truth expression is checked at the end of each iteration instead of in the beginning. The main difference from regular `while` loops is that the first iteration of a `do..while` loop is guaranteed to run (the truth expression is only checked at the end of the iteration), whereas it's may not necessarily run with a regular `while` loop (the truth expression is checked at the beginning of each iteration, if it evaluates to `FALSE` right from the beginning, the loop execution would end immediately).

There is just one syntax for `do..while` loops:

```
$i = 0;
do {
   print $i;
} while ($i>0);
```

The above loop would run one time exactly, since after the first iteration, when truth expression is checked, it evaluates to `FALSE` ($i is not bigger than 0) and the loop execution ends.

Advanced C users may be familiar with a different usage of the `do..while` loop, to allow stopping execution in the middle of code blocks, by encapsulating them with `do..while`(0), and using the `break` statement. The following code fragment demonstrates this:

```
do {
    if ($i < 5) {
        print "i is not big enough";
        break;
    }
    $i *= $factor;
    if ($i < $minimum_limit) {
        break;
    }
    print "i is ok";

     ...process i...

} while(0);
```

Don’t worry if you don’t understand this right away or at all. You can code scripts and even powerful scripts without using this ’feature’.

for

`for` loops are the most complex loops in PHP. They behave like their C counterparts. The syntax of a `for` loop is:

```
for (expr1; expr2; expr3) statement
```

The first expression (*`expr1`*) is evaluated (executed) once unconditionally at the beginning of the loop.

In the beginning of each iteration, *`expr2`* is evaluated. If it evaluates to `TRUE`, the loop continues and the nested statement(s) are executed. If it evaluates to `FALSE`, the execution of the loop ends.

At the end of each iteration, *`expr3`* is evaluated (executed).

Each of the expressions can be empty. *`expr2`* being empty means the loop should be run indefinitely (PHP implicitly considers it as `TRUE`, like C). This may not be as useless as you might think, since often you'd want to end the loop using a conditional `break` statement instead of using the `for` truth expression.

Consider the following examples. All of them display numbers from 1 to 10:

```
/* example 1 */

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

/* example 2 */

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

/* example 3 */

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

/* example 4 */

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

Of course, the first example appears to be the nicest one (or perhaps the fourth), but you may find that being able to use empty expressions in `for` loops comes in handy in many occasions.

PHP also supports the alternate "colon syntax" for `for` loops.

```
for (expr1; expr2; expr3): statement; ...; endfor;
```

Other languages have a `foreach` statement to traverse an array or hash. PHP 3 has no such construct; PHP 4 does (see foreach). In PHP 3, you can combine while with the list() and each() functions to achieve the same effect. See the documentation for these functions for an example.

foreach

PHP 4 (not PHP 3) includes a `foreach` construct, much like Perl and some other languages. This simply gives an easy way to iterate over arrays. `foreach` works only on arrays, and will issue an error when you try to use it on a variable with a different data type or an uninitialized variables. There are two syntaxes; the second is a minor but useful extension of the first:

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

The first form loops over the array given by `array_expression`. On each loop, the value of the current element is assigned to `$value` and the internal array pointer is advanced by one (so on the next loop, you'll be looking at the next element).

The second form does the same thing, except that the current element's key will be assigned to the variable `$key` on each loop.

> **Note:** When `foreach` first starts executing, the internal array pointer is automatically reset to the first element of the array. This means that you do not need to call reset() before a `foreach` loop.

> **Note:** Also note that `foreach` operates on a copy of the specified array, not the array itself, therefore the array pointer is not modified as with the each() construct and changes to the array element returned are not reflected in the original array.

> **Note:** `foreach` does not support the ability to suppress error messages using '@'.

You may have noticed that the following are functionally identical:

```
reset ($arr);
while (list(, $value) = each ($arr)) {
    echo "Value: $value<br>\n";
}

foreach ($arr as $value) {
    echo "Value: $value<br>\n";
}
```

The following are also functionally identical:

```
reset ($arr);
while (list($key, $value) = each ($arr)) {
    echo "Key: $key; Value: $value<br>\n";
}

foreach ($arr as $key => $value) {
    echo "Key: $key; Value: $value<br>\n";
}
```

Some more examples to demonstrate usages:

```
/* foreach example 1: value only */

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

foreach ($a as $v) {
   print "Current value of \$a: $v.\n";
}

/* foreach example 2: value (with key printed for illustration) */

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

$i = 0; /* for illustrative purposes only */

foreach($a as $v) {
    print "\$a[$i] => $v.\n";
    $i++;
}
```

```
/* foreach example 3: key and value */

$a = array (
    "one" => 1,
    "two" => 2,
    "three" => 3,
    "seventeen" => 17
);

foreach($a as $k => $v) {
    print "\$a[$k] => $v.\n";
}

/* foreach example 4: multi-dimensional arrays */

$a[0][0] = "a";
$a[0][1] = "b";
$a[1][0] = "y";
$a[1][1] = "z";

foreach($a as $v1) {
    foreach ($v1 as $v2) {
        print "$v2\n";
    }
}

/* foreach example 5: dynamic arrays */

foreach(array(1, 2, 3, 4, 5) as $v) {
    print "$v\n";
}
```

break

`break` ends execution of the current `for`, `foreach` `while`, `do..while` or `switch` structure.

`break` accepts an optional numeric argument which tells it how many nested enclosing structures are to be broken out of.

```
$arr = array ('one', 'two', 'three', 'four', 'stop', 'five');
while (list (, $val) = each ($arr)) {
    if ($val == 'stop') {
        break;    /* You could also write 'break 1;' here. */
    }
    echo "$val<br>\n";
}
```

```
/* Using the optional argument. */

$i = 0;
while (++$i) {
    switch ($i) {
    case 5:
        echo "At 5<br>\n";
        break 1;  /* Exit only the switch. */
    case 10:
        echo "At 10; quitting<br>\n";
        break 2;  /* Exit the switch and the while. */
    default:
        break;
    }
}
```

continue

`continue` is used within looping structures to skip the rest of the current loop iteration and continue execution at the beginning of the next iteration.

`continue` accepts an optional numeric argument which tells it how many levels of enclosing loops it should skip to the end of.

```
while (list ($key, $value) = each ($arr)) {
    if (!($key % 2)) { // skip odd members
        continue;
    }
    do_something_odd ($value);
}

$i = 0;
while ($i++ < 5) {
    echo "Outer<br>\n";
    while (1) {
        echo "&nbsp;&nbsp;Middle<br>\n";
        while (1) {
            echo "&nbsp;&nbsp;Inner<br>\n";
            continue 3;
        }
        echo "This never gets output.<br>\n";
    }
    echo "Neither does this.<br>\n";
}
```

switch

The `switch` statement is similar to a series of IF statements on the same expression. In many occasions, you may want to compare the same variable (or expression) with many different values, and execute a different piece of code depending on which value it equals to. This is exactly what the `switch` statement is for.

The following two examples are two different ways to write the same thing, one using a series of `if` statements, and the other using the `switch` statement:

```
if ($i == 0) {
    print "i equals 0";
}
if ($i == 1) {
    print "i equals 1";
}
if ($i == 2) {
    print "i equals 2";
}

switch ($i) {
    case 0:
        print "i equals 0";
        break;
    case 1:
        print "i equals 1";
        break;
    case 2:
        print "i equals 2";
        break;
}
```

It is important to understand how the `switch` statement is executed in order to avoid mistakes. The `switch` statement executes line by line (actually, statement by statement). In the beginning, no code is executed. Only when a `case` statement is found with a value that matches the value of the `switch` expression does PHP begin to execute the statements. PHP continues to execute the statements until the end of the `switch` block, or the first time it sees a `break` statement. If you don’t write a `break` statement at the end of a case’s statement list, PHP will go on executing the statements of the following case. For example:

```
switch ($i) {
    case 0:
        print "i equals 0";
    case 1:
        print "i equals 1";
    case 2:
        print "i equals 2";
}
```

Here, if $i is equal to 0, PHP would execute all of the print statements! If $i is equal to 1, PHP would execute the last two print statements. You would get the expected behavior ('i equals 2' would be displayed) only if $i is equal to 2. Thus, it is important not to forget `break` statements (even though you may want to avoid supplying them on purpose under certain circumstances).

In a `switch` statement, the condition is evaluated only once and the result is compared to each `case` statement. In an `elseif` statement, the condition is evaluated again. If your condition is more complicated than a simple compare and/or is in a tight loop, a `switch` may be faster.

The statement list for a case can also be empty, which simply passes control into the statement list for the next case.

```
switch ($i) {
    case 0:
    case 1:
    case 2:
        print "i is less than 3 but not negative";
        break;
    case 3:
        print "i is 3";
}
```

A special case is the default case. This case matches anything that wasn't matched by the other cases, and should be the last `case` statement. For example:

```
switch ($i) {
    case 0:
        print "i equals 0";
        break;
    case 1:
        print "i equals 1";
        break;
    case 2:
        print "i equals 2";
        break;
    default:
        print "i is not equal to 0, 1 or 2";
}
```

The `case` expression may be any expression that evaluates to a simple type, that is, integer or floating-point numbers and strings. Arrays or objects cannot be used here unless they are dereferenced to a simple type.

The alternative syntax for control structures is supported with switches. For more information, see Alternative syntax for control structures .

```
switch ($i):
    case 0:
        print "i equals 0";
        break;
    case 1:
        print "i equals 1";
        break;
    case 2:
        print "i equals 2";
        break;
    default:
        print "i is not equal to 0, 1 or 2";
endswitch;
```

declare

The `declare` construct is used to set execution directives for a block of code. The syntax of `declare` is similar to the syntax of other flow control constructs:

```
declare (directive) statement
```

The `directive` section allows the behavior of the `declare` block to be set. Currently only one directive is recognized: the `ticks` directive. (See below for more information on the ticks directive)

The `statement` part of the `declare` block will be executed -- how it is executed and what side effects occur during execution may depend on the directive set in the `directive` block.

Ticks

A tick is an event that occurs for every `N` low-level statements executed by the parser within the `declare` block. The value for `N` is specified using `ticks=N` within the `declare` blocks's `directive` section.

The event(s) that occur on each tick are specified using the register_tick_function(). See the example below for more details. Note that more than one event can occur for each tick.

Example 12-1. Profile a section of PHP code

```
<?php
// A function that records the time when it is called
function profile ($dump = FALSE)
{
    static $profile;

    // Return the times stored in profile, then erase it
    if ($dump) {
        $temp = $profile;
        unset ($profile);
        return ($temp);
    }

    $profile[] = microtime ();
}

// Set up a tick handler
register_tick_function("profile");

// Initialize the function before the declare block
profile ();

// Run a block of code, throw a tick every 2nd statement
declare (ticks=2) {
    for ($x = 1; $x < 50; ++$x) {
        echo similar_text (md5($x), md5($x*$x)), "<br />;";
    }
}

// Display the data stored in the profiler
print_r (profile (TRUE));
?>
```

The example profiles the PHP code within the ’declare’ block, recording the time at which every second low-level statement in the block was executed. This information can then be used to find the slow areas within particular segments of code. This process can be performed using other methods: using ticks is more convenient and easier to implement.

Ticks are well suited for debugging, implementing simple multitasking, backgrounded I/O and many other tasks.

See also register_tick_function() and unregister_tick_function().

return

If called from within a function, the return() statement immediately ends execution of the current function, and returns its argument as the value of the function call. return() will also end the execution of an eval() statement or script file.

If called from the global scope, then execution of the current script file is ended. If the current script file was include()ed or require()ed, then control is passed back to the calling file. Furthermore, if the current script file was include()ed, then the value given to return() will be returned as the value of the include() call. If return() is called from within the main script file, then script execution ends. If the current script file was named by the auto_prepend_file or auto_append_file configuration options in the configuration file, then that script file's execution is ended.

For more information, see Returning values.

> **Note:** Note that since return() is a language construct and not a function, the parentheses surrounding its arguments are *not* required--in fact, it is more common to leave them out than to use them, although it doesn't matter one way or the other.

require()

The require() statement includes and evaluates the specific file.

require() includes and evaluates a specific file. Detailed information on how this inclusion works is described in the documentation for include().

require() and include() are identical in every way except how they handle failure. include() produces a Warning while require() results in a Fatal Error. In other words, don't hesitate to use require() if you want a missing file to halt processing of the page. include() does not behave this way, the script will continue regardless. Be sure to have an appropriate include_path setting as well.

Example 12-2. Basic require() examples

```
<?php

require ’prepend.php’;

require $somefile;

require (’somefile.txt’);

?>
```

See the include() documentation for more examples.

> **Note:** Prior to PHP 4.0.2, the following applies: require() will always attempt to read the target file, even if the line it's on never executes. The conditional statement won't affect require(). However, if the line on which the require() occurs is not executed, neither will any of the code in the target file be executed. Similarly, looping structures do not affect the behaviour of require(). Although the code contained in the target file is still subject to the loop, the require() itself happens only once.

> **Warning**
>
> Windows versions of PHP prior to PHP 4.3 do not support accessing remote files via this function, even if allow_url_fopen is enabled.

See also include(), require_once(), include_once(), eval(), file(), readfile(), virtual() and include_path.

include()

The include() statement includes and evaluates the specified file.

The documentation below also applies to require(). The two constructs are identical in every way except how they handle failure. include() produces a Warning while require() results in a Fatal Error. In other words, use require() if you want a missing file to halt processing of the page. include() does not behave this way, the script will continue regardless. Be sure to have an appropriate include_path setting as well.

When a file is included, the code it contains inherits the variable scope of the line on which the include occurs. Any variables available at that line in the calling file will be available within the called file, from that point forward.

Example 12-3. Basic include() example

```
vars.php
<?php

$color = 'green';
$fruit = 'apple';

?>

test.php
<?php

echo "A $color $fruit"; // A

include 'vars.php';

echo "A $color $fruit"; // A green apple
```

```
?>
```

If the include occurs inside a function within the calling file, then all of the code contained in the called file will behave as though it had been defined inside that function. So, it will follow the variable scope of that function.

Example 12-4. Including within functions

```
<?php

function foo()
{
global $color;

    include ’vars.php’;

    echo "A $color $fruit";
}

/* vars.php is in the scope of foo() so     *
 * $fruit is NOT available outside of this  *
 * scope.  $color is because we declared it *
 * as global.                               */

foo();                    // A green apple
echo "A $color $fruit";   // A green

?>
```

When a file is included, parsing drops out of PHP mode and into HTML mode at the beginning of the target file, and resumes again at the end. For this reason, any code inside the target file which should be executed as PHP code must be enclosed within valid PHP start and end tags.

If "URL fopen wrappers" are enabled in PHP (which they are in the default configuration), you can specify the file to be included using an URL (via HTTP or other supported wrapper - see Appendix I for a list of protocols) instead of a local pathname. If the target server interprets the target file as PHP code, variables may be passed to the included file using an URL request string as used with HTTP GET. This is not strictly speaking the same thing as including the file and having it inherit the parent file’s variable scope; the script is actually being run on the remote server and the result is then being included into the local script.

Warning

Windows versions of PHP prior to PHP 4.3 do not support accessing remote files via this function, even if allow_url_fopen is enabled.

Example 12-5. include() through HTTP

```
<?php

/* This example assumes that www.example.com is configured to parse .php *
 * files and not .txt files. Also, 'Works' here means that the variables *
 * $foo and $bar are available within the included file.                 */

// Won't work; file.txt wasn't handled by www.example.com as PHP
include 'http://www.example.com/file.txt?foo=1&bar=2';

// Won't work; looks for a file named 'file.php?foo=1&bar=2' on the
// local filesystem.
include 'file.php?foo=1&bar=2';

// Works.
include 'http://www.example.com/file.php?foo=1&bar=2';

$foo = 1;
$bar = 2;
include 'file.txt';  // Works.
include 'file.php';  // Works.

?>
```

See also Remote files, fopen() and file() for related information.

Because include() and require() are special language constructs, you must enclose them within a statement block if it's inside a conditional block.

Example 12-6. include() and conditional blocks

```
<?php

// This is WRONG and will not work as desired.
if ($condition)
    include $file;
else
    include $other;
```

```
// This is CORRECT.
if ($condition) {
    include $file;
} else {
    include $other;
}

?>
```

Handling Returns: It is possible to execute a return() statement inside an included file in order to terminate processing in that file and return to the script which called it. Also, it’s possible to return values from included files. You can take the value of the include call as you would a normal function.

> **Note:** In PHP 3, the return may not appear inside a block unless it’s a function block, in which case the return() applies to that function and not the whole file.

Example 12-7. include() and the return() statement

```
return.php
<?php

$var = ’PHP’;

return $var;

?>

noreturn.php
<?php

$var = ’PHP’;

?>

testreturns.php
<?php

$foo = include ’return.php’;

echo $foo; // prints ’PHP’

$bar = include ’noreturn.php’;

echo $bar; // prints 1
```

```
?>
```

`$bar` is the value `1` because the include was successful. Notice the difference between the above examples. The first uses return() within the included file while the other does not. A few other ways to "include" files into variables are with fopen(), file() or by using include() along with Output Control Functions.

See also require(), require_once(), include_once(), readfile(), virtual(), and include_path.

require_once()

The require_once() statement includes and evaluates the specified file during the execution of the script. This is a behavior similar to the require() statement, with the only difference being that if the code from a file has already been included, it will not be included again. See the documentation for require() for more information on how this statement works.

require_once() should be used in cases where the same file might be included and evaluated more than once during a particular execution of a script, and you want to be sure that it is included exactly once to avoid problems with function redefinitions, variable value reassignments, etc.

For examples on using require_once() and include_once(), look at the PEAR (http://pear.php.net/) code included in the latest PHP source code distributions.

Note: require_once() was added in PHP 4.0.1pl2

Note: Be aware, that the behaviour of require_once() and include_once() may not be what you expect on a non case sensitive operating system (such as Windows).

Example 12-8. require_once() is case sensitive

```
require_once("a.php"); // this will include a.php
require_once("A.php"); // this will include a.php again on Windows!
```

Warning

Windows versions of PHP prior to PHP 4.3 do not support accessing remote files via this function, even if allow_url_fopen is enabled.

See also: require(), include(), include_once(), get_required_files(), get_included_files(), readfile(), and virtual().

include_once()

The include_once() statement includes and evaluates the specified file during the execution of the script. This is a behavior similar to the include() statement, with the only difference being that if the code from a file has already been included, it will not be included again. As the name suggests, it will be included just once.

include_once() should be used in cases where the same file might be included and evaluated more than once during a particular execution of a script, and you want to be sure that it is included exactly once to avoid problems with function redefinitions, variable value reassignments, etc.

For more examples on using require_once() and include_once(), look at the PEAR (http://pear.php.net/) code included in the latest PHP source code distributions.

Note: include_once() was added in PHP 4.0.1pl2

Note: Be aware, that the behaviour of include_once() and require_once() may not be what you expect on a non case sensitive operating system (such as Windows).

Example 12-9. include_once() is case sensitive

```
include_once("a.php"); // this will include a.php
include_once("A.php"); // this will include a.php again on Windows!
```

Warning

Windows versions of PHP prior to PHP 4.3 do not support accessing remote files via this function, even if allow_url_fopen is enabled.

See also include(), require(), require_once(), get_required_files(), get_included_files(), readfile(), and virtual().

Chapter 13. Functions

User-defined functions

A function may be defined using syntax such as the following:

```
function foo ($arg_1, $arg_2, ..., $arg_n)
{
    echo "Example function.\n";
    return $retval;
}
```

Any valid PHP code may appear inside a function, even other functions and class definitions.

In PHP 3, functions must be defined before they are referenced. No such requirement exists in PHP 4.

PHP does not support function overloading, nor is it possible to undefine or redefine previously-declared functions.

PHP 3 does not support variable numbers of arguments to functions, although default arguments are supported (see Default argument values for more information). PHP 4 supports both: see Variable-length argument lists and the function references for func_num_args(), func_get_arg(), and func_get_args() for more information.

Function arguments

Information may be passed to functions via the argument list, which is a comma-delimited list of variables and/or constants.

PHP supports passing arguments by value (the default), passing by reference, and default argument values. Variable-length argument lists are supported only in PHP 4 and later; see Variable-length argument lists and the function references for func_num_args(), func_get_arg(), and func_get_args() for more information. A similar effect can be achieved in PHP 3 by passing an array of arguments to a function:

```
function takes_array($input)
{
    echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
```

Making arguments be passed by reference

By default, function arguments are passed by value (so that if you change the value of the argument within the function, it does not get changed outside of the function). If you wish to allow a function to modify its arguments, you must pass them by reference.

If you want an argument to a function to always be passed by reference, you can prepend an ampersand (&) to the argument name in the function definition:

```
function add_some_extra(&$string)
{
    $string .= ’and something extra.’;
}
$str = ’This is a string, ’;
add_some_extra($str);
echo $str;    // outputs ’This is a string, and something extra.’
```

Default argument values

A function may define C++-style default values for scalar arguments as follows:

```
function makecoffee ($type = "cappuccino")
{
    return "Making a cup of $type.\n";
}
echo makecoffee ();
echo makecoffee ("espresso");
```

The output from the above snippet is:

```
Making a cup of cappuccino.
Making a cup of espresso.
```

The default value must be a constant expression, not (for example) a variable or class member.

Note that when using default arguments, any defaults should be on the right side of any non-default arguments; otherwise, things will not work as expected. Consider the following code snippet:

```
function makeyogurt ($type = "acidophilus", $flavour)
```

```
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt ("raspberry");   // won’t work as expected
```

The output of the above example is:

```
Warning: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/php3test/functest.html on line 41
Making a bowl of raspberry .
```

Now, compare the above with this:

```
function makeyogurt ($flavour, $type = "acidophilus")
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt ("raspberry");   // works as expected
```

The output of this example is:

```
Making a bowl of acidophilus raspberry.
```

Variable-length argument lists

PHP 4 has support for variable-length argument lists in user-defined functions. This is really quite easy, using the func_num_args(), func_get_arg(), and func_get_args() functions.

No special syntax is required, and argument lists may still be explicitly provided with function definitions and will behave as normal.

Returning values

Values are returned by using the optional return statement. Any type may be returned, including lists and objects. This causes the function to end its execution immediately and pass control back to the line from which it was called. See return() for more information.

```
function square ($num)
{
    return $num * $num;
}
echo square (4);   // outputs ’16’.
```

You can’t return multiple values from a function, but similar results can be obtained by returning a list.

```
function small_numbers()
{
    return array (0, 1, 2);
}
list ($zero, $one, $two) = small_numbers();
```

To return a reference from a function, you have to use the reference operator & in both the function declaration and when assigning the returned value to a variable:

```
function &returns_reference()
{
    return $someref;
}

$newref =& returns_reference();
```

For more information on references, please check out References Explained.

`old_function`

The `old_function` statement allows you to declare a function using a syntax identical to PHP/FI2 (except you must replace ’function’ with ’old_function’.

This is a deprecated feature, and should only be used by the PHP/FI2->PHP 3 convertor.

Warning

Functions declared as `old_function` cannot be called from PHP's internal code. Among other things, this means you can't use them in functions such as usort(), array_walk(), and register_shutdown_function(). You can get around this limitation by writing a wrapper function (in normal PHP 3 form) to call the `old_function`.

Variable functions

PHP supports the concept of variable functions. This means that if a variable name has parentheses appended to it, PHP will look for a function with the same name as whatever the variable evaluates to, and will attempt to execute it. Among other things, this can be used to implement callbacks, function tables, and so forth.

Variable functions won't work with language constructs such as echo(), unset(), isset(), empty(), include() and print().

Example 13-1. Variable function example

```
<?php
function foo()
{
    echo "In foo()<br>\n";
}

function bar($arg = ”)
{
    echo "In bar(); argument was ’$arg’.<br>\n";
}

$func = ’foo’;
$func();
$func = ’bar’;
$func(’test’);
?>
```

See also variable variables and function_exists().

Chapter 14. Classes and Objects

class

A class is a collection of variables and functions working with these variables. A class is defined using the following syntax:

```
<?php
class Cart
{
    var $items;  // Items in our shopping cart

    // Add $num articles of $artnr to the cart

    function add_item ($artnr, $num)
    {
        $this->items[$artnr] += $num;
    }

    // Take $num articles of $artnr out of the cart

    function remove_item ($artnr, $num)
    {
        if ($this->items[$artnr] > $num) {
            $this->items[$artnr] -= $num;
            return true;
        } else {
            return false;
        }
    }
}
?>
```

This defines a class named Cart that consists of an associative array of articles in the cart and two functions to add and remove items from this cart.

Caution

The following cautionary notes are valid for PHP 4.

The name `stdClass` is used interally by Zend and is reserved. You cannot have a class named `stdClass` in PHP.

The function names `__sleep` and `__wakeup` are magical in PHP classes. You cannot have functions with these names in any of your classes unless you want the magic functionality associated with them. See below for more information.

PHP reserves all function names starting with __ as magical. It is recommended that you do not use function names with __ in PHP unless you want some documented magic functionality.

Note: In PHP 4, only constant initializers for `var` variables are allowed. To initialize variables with non-constant values, you need an initialization function which is called automatically when an object is being constructed from the class. Such a function is called a constructor (see below).

```
<?php
/* None of these will work in PHP 4. */
class Cart
{
    var $todays_date = date("Y-m-d");
    var $name = $firstname;
    var $owner = ’Fred ’ . ’Jones’;
    var $items = array("VCR", "TV");
}

/* This is how it should be done. */
class Cart
{
    var $todays_date;
    var $name;
    var $owner;
    var $items;

    function Cart()
    {
        $this->todays_date = date("Y-m-d");
        $this->name = $GLOBALS[’firstname’];
        /* etc. . . */
    }
}
?>
```

Classes are types, that is, they are blueprints for actual variables. You have to create a variable of the desired type with the `new` operator.

```
<?php
$cart = new Cart;
$cart->add_item("10", 1);

$another_cart = new Cart;
$another_cart->add_item("0815", 3);
```

This creates the objects $cart and $another_cart, both of the class Cart. The function add_item() of the $cart object is being called to add 1 item of article number 10 to the $cart. 3 items of article number 0815 are being added to $another_cart.

Both, $cart and $another_cart, have functions add_item(), remove_item() and a variable items. These are distinct functions and variables. You can think of the objects as something similar to directories in a filesystem. In a filesystem you can have two different files README.TXT, as long as they are in different directories. Just like with directories where you'll have to type the full pathname in order to reach each file from the toplevel directory, you have to specify the complete name of the function you want to call: In PHP terms, the toplevel directory would be the global namespace, and the pathname separator would be ->. Thus, the names $cart->items and $another_cart->items name two different variables. Note that the variable is named $cart->items, not $cart->$items, that is, a variable name in PHP has only a single dollar sign.

```
// correct, single $
$cart->items = array("10" => 1);

// invalid, because $cart->$items becomes $cart->""
$cart->$items = array("10" => 1);

// correct, but may or may not be what was intended:
// $cart->$myvar becomes $cart->items
$myvar = 'items';
$cart->$myvar = array("10" => 1);
```

Within a class definition, you do not know under which name the object will be accessible in your program: at the time the Cart class was written, it was unknown that the object will be named $cart or $another_cart later. Thus, you cannot write $cart->items within the Cart class itself. Instead, in order to be able to access it's own functions and variables from within a class, one can use the pseudo-variable $this which can be read as 'my own' or 'current object'. Thus, '$this->items[$artnr] += $num' can be read as 'add $num to the $artnr counter of my own items array' or 'add $num to the $artnr counter of the items array within the current object'.

Note: There are some nice functions to handle classes and objects. You might want to take a look at the Class/Object Functions

extends

Often you need classes with similar variables and functions to another existing class. In fact, it is good practice to define a generic class which can be used in all your projects and adapt this class for the needs of each of your specific projects. To facilitate this, classes can be extensions of other classes. The extended or derived class has all variables and functions of the base class (this is called 'inheritance' despite the fact that nobody died) and what you add in the extended definition. It is not possible to substract from a class, that is, to undefine any existing functions or variables. An extended class is always dependent on a single base class, that is, multiple inheritance is not supported. Classes are extended using the keyword 'extends'.

```
class Named_Cart extends Cart
{
    var $owner;

    function set_owner ($name)
    {
        $this->owner = $name;
    }
}
```

This defines a class Named_Cart that has all variables and functions of Cart plus an additional variable $owner and an additional function set_owner(). You create a named cart the usual way and can now set and get the carts owner. You can still use normal cart functions on named carts:

```
$ncart = new Named_Cart;    // Create a named cart
$ncart->set_owner("kris");  // Name that cart
print $ncart->owner;        // print the cart owners name
$ncart->add_item("10", 1);  // (inherited functionality from cart)
```

This is also called a "parent-child" relationship. You create a class, parent, and use `extends` to create a new class *based* on the parent class: the child class. You can even use this new child class and create another class based on this child class.

> **Note:** Classes must be defined before they are used! If you want the class `Named_Cart` to extend the class `Cart`, you will have to define the class `Cart` first. If you want to create another class called `Yellow_named_cart` based on the class `Named_Cart` you have to define `Named_Cart` first. To make it short: the order in which the classes are defined is important.

Constructors

Caution

In PHP 3 and PHP 4 constructors behave differently. The PHP 4 semantics are strongly preferred.

Constructors are functions in a class that are automatically called when you create a new instance of a class with `new`. In PHP 3, a function becomes a constructor when it has the same name as the class. In PHP 4, a function becomes a constructor, when it has the same name as the class it is defined in - the difference is subtle, but crucial (see below).

```
// Works in PHP 3 and PHP 4.
class Auto_Cart extends Cart
{
    function Auto_Cart()
    {
        $this->add_item ("10", 1);
    }
}
```

This defines a class Auto_Cart that is a Cart plus a constructor which initializes the cart with one item of article number "10" each time a new Auto_Cart is being made with "new". Constructors can take arguments and these arguments can be optional, which makes them much more useful. To be able to still use the class without parameters, all parameters to constructors should be made optional by providing default values.

```
// Works in PHP 3 and PHP 4.
class Constructor_Cart extends Cart
{
    function Constructor_Cart($item = "10", $num = 1)
    {
        $this->add_item ($item, $num);
    }
}

// Shop the same old boring stuff.

$default_cart = new Constructor_Cart;

// Shop for real...

$different_cart = new Constructor_Cart("20", 17);
```

You also can use the `@` operator to *mute* errors occuring in the constructor, e.g. `@new`.

Caution

In PHP 3, derived classes and constructors have a number of limitations. The following examples should be read carefully to understand these limitations.

```
class A
{
    function A()
    {
      echo "I am the constructor of A.<br>\n";
    }
}

class B extends A
```

```
{
    function C()
    {
        echo "I am a regular function.<br>\n";
    }
}

// no constructor is being called in PHP 3.
$b = new B;
```

In PHP 3, no constructor is being called in the above example. The rule in PHP 3 is: ’A constructor is a function of the same name as the class.’. The name of the class is B, and there is no function called B() in class B. Nothing happens.

This is fixed in PHP 4 by introducing another rule: If a class has no constructor, the constructor of the base class is being called, if it exists. The above example would have printed ’I am the constructor of A.<br>’ in PHP 4.

```
class A
{
    function A()
    {
        echo "I am the constructor of A.<br>\n";
    }

    function B()
    {
        echo "I am a regular function named B in class A.<br>\n";
        echo "I am not a constructor in A.<br>\n";
    }
}

class B extends A
{
    function C()
    {
        echo "I am a regular function.<br>\n";
    }
}

// This will call B() as a constructor.
$b = new B;
```

In PHP 3, the function B() in class A will suddenly become a constructor in class B, although it was never intended to be. The rule in PHP 3 is: ’A constructor is a function of the same name as the class.’. PHP 3 does not care if the function is being defined in class B, or if it has been inherited.

This is fixed in PHP 4 by modifying the rule to: ’A constructor is a function of the same name as the class it is being defined in.’. Thus in PHP 4, the class B would have no constructor function of its own and the constructor of the base class would have been called, printing ’I am the constructor of A.<br>’.

Caution

Neither PHP 3 nor PHP 4 call constructors of the base class automatically from a constructor of a derived class. It is your responsibility to propagate the call to constructors upstream where appropriate.

Note: There are no destructors in PHP 3 or PHP 4. You may use register_shutdown_function() instead to simulate most effects of destructors.

Destructors are functions that are called automatically when an object is destroyed, either with unset() or by simply going out of scope. There are no destructors in PHP.

::

Caution

The following is valid for PHP 4 only.

Sometimes it is useful to refer to functions and variables in base classes or to refer to functions in classes that have not yet any instances. The :: operator is being used for this.

```
class A
{
    function example()
    {
        echo "I am the original function A::example().<br>\n";
    }
}

class B extends A
{
    function example()
    {
        echo "I am the redefined function B::example().<br>\n";
        A::example();
    }
}

// there is no object of class A.
// this will print
//   I am the original function A::example().<br>
A::example();
```

```
// create an object of class B.
$b = new B;

// this will print
//   I am the redefined function B::example().<br>
//   I am the original function A::example().<br>
$b->example();
```

The above example calls the function example() in class A, but there is no object of class A, so that we cannot write $a->example() or similar. Instead we call example() as a ’class function’, that is, as a function of the class itself, not any object of that class.

There are class functions, but there are no class variables. In fact, there is no object at all at the time of the call. Thus, a class function may not use any object variables (but it can use local and global variables), and it may no use $this at all.

In the above example, class B redefines the function example(). The original definition in class A is shadowed and no longer available, unless you are refering specifically to the implementation of example() in class A using the ::-operator. Write A::example() to do this (in fact, you should be writing parent::example(), as shown in the next section).

In this context, there is a current object and it may have object variables. Thus, when used from WITHIN an object function, you may use $this and object variables.

parent

You may find yourself writing code that refers to variables and functions in base classes. This is particularly true if your derived class is a refinement or specialisation of code in your base class.

Instead of using the literal name of the base class in your code, you should be using the special name `parent`, which refers to the name of your base class as given in the `extends` declaration of your class. By doing this, you avoid using the name of your base class in more than one place. Should your inheritance tree change during implementation, the change is easily made by simply changing the `extends` declaration of your class.

```
class A
{
    function example()
    {
        echo "I am A::example() and provide basic functionality.<br>\n";
    }
}

class B extends A
{
    function example()
    {
        echo "I am B::example() and provide additional functionality.<br>\n";
        parent::example();
    }
}

$b = new B;

// This will call B::example(), which will in turn call A::example().
$b->example();
```

Serializing objects - objects in sessions

> **Note:** In PHP 3, objects will lose their class association throughout the process of serialization and unserialization. The resulting variable is of type object, but has no class and no methods, thus it is pretty useless (it has become just like an array with a funny syntax).

Caution

The following information is valid for PHP 4 only.

serialize() returns a string containing a byte-stream representation of any value that can be stored in PHP. unserialize() can use this string to recreate the original variable values. Using serialize to save an object will save all variables in an object. The functions in an object will not be saved, only the name of the class.

In order to be able to unserialize() an object, the class of that object needs to be defined. That is, if you have an object $a of class A on page1.php and serialize this, you'll get a string that refers to class A and contains all values of variabled contained in $a. If you want to be able to unserialize this on page2.php, recreating $a of class A, the definition of class A must be present in page2.php. This can be done for example by storing the class defintion of class A in an include file and including this file in both page1.php and page2.php.

```
classa.inc:
  class A
  {
      var $one = 1;

      function show_one()
      {
          echo $this->one;
      }
  }

page1.php:
  include("classa.inc");
  $a = new A;
  $s = serialize($a);
  // store $s somewhere where page2.php can find it.
  $fp = fopen("store", "w");
  fputs($fp, $s);
  fclose($fp);

page2.php:
  // this is needed for the unserialize to work properly.
  include("classa.inc");

  $s = implode("", @file("store"));
  $a = unserialize($s);

  // now use the function show_one() of the $a object.
  $a->show_one();
```

If you are using sessions and use session_register() to register objects, these objects are serialized automatically at the end of each PHP page, and are unserialized automatically on each of the following pages. This basically means that these objects can show up on any of your pages once they become part of your session.

It is strongly recommended that you include the class definitions of all such registered objects on all of your pages, even if you do not actually use these classes on all of your pages. If you don't and an object is being unserialized without its class definition being present, it will lose its class association and become an object of class `stdClass` without any functions available at all, that is, it will become quite useless.

So if in the example above $a became part of a session by running `session_register("a")`, you should include the file `classa.inc` on all of your pages, not only page1.php and page2.php.

The magic functions `__sleep` and `__wakeup`

serialize() checks if your class has a function with the magic name `__sleep`. If so, that function is being run prior to any serialization. It can clean up the object and is supposed to return an array with the names of all variables of that object that should be serialized.

The intended use of `__sleep` is to close any database connections that object may have, committing pending data or perform similar cleanup tasks. Also, the function is useful if you have very large objects which need not be saved completely.

Conversely, unserialize() checks for the presence of a function with the magic name `__wakeup`. If present, this function can reconstruct any resources that object may have.

The intended use of `__wakeup` is to reestablish any database connections that may have been lost during serialization and perform other reinitialization tasks.

References inside the constructor

Creating references within the constructor can lead to confusing results. This tutorial-like section helps you to avoid problems.

```
class Foo
{
    function Foo($name)
    {
        // create a reference inside the global array $globalref
        global $globalref;
        $globalref[] = &$this;
        // set name to passed value
        $this->setName($name);
        // and put it out
        $this->echoName();
    }

    function echoName()
    {
        echo "<br>",$this->name;
    }

    function setName($name)
    {
        $this->name = $name;
    }
}
```

Let us check out if there is a difference between `$bar1` which has been created using the copy = operator and `$bar2` which has been created using the reference =& operator...

```
$bar1 = new Foo('set in constructor');
$bar1->echoName();
$globalref[0]->echoName();

/* output:
set in constructor
set in constructor
set in constructor */

$bar2 =& new Foo('set in constructor');
$bar2->echoName();
$globalref[1]->echoName();

/* output:
set in constructor
set in constructor
set in constructor */
```

Apparently there is no difference, but in fact there is a very significant one: `$bar1` and `$globalref[0]` are _NOT_ referenced, they are NOT the same variable. This is because "new" does not return a reference by default, instead it returns a copy.

> **Note:** There is no performance loss (since PHP 4 and up use reference counting) returning copies instead of references. On the contrary it is most often better to simply work with copies instead of references, because creating references takes some time where creating copies virtually takes no time (unless none of them is a large array or object and one of them gets changed and the other(s) one(s) subsequently, then it would be wise to use references to change them all concurrently).

To prove what is written above let us watch the code below.

```
// now we will change the name. what do you expect?
// you could expect that both $bar1 and $globalref[0] change their names...
$bar1->setName('set from outside');

// as mentioned before this is not the case.
$bar1->echoName();
$globalref[0]->echoName();

/* output:
set from outside
set in constructor */

// let us see what is different with $bar2 and $globalref[1]
$bar2->setName('set from outside');

// luckily they are not only equal, they are the same variable
// thus $bar2->name and $globalref[1]->name are the same too
$bar2->echoName();
$globalref[1]->echoName();

/* output:
set from outside
set from outside */
```

Another final example, try to understand it.

```
class A
{
    function A($i)
    {
        $this->value = $i;
        // try to figure out why we do not need a reference here
        $this->b = new B($this);
    }

    function createRef()
    {
        $this->c = new B($this);
    }

    function echoValue()
    {
        echo "<br>","class ",get_class($this),’: ’,$this->value;
    }
}


class B
{
    function B(&$a)
    {
        $this->a = &$a;
    }

    function echoValue()
    {
        echo "<br>","class ",get_class($this),’: ’,$this->a->value;
    }
}

// try to undestand why using a simple copy here would yield
// in an undesired result in the *-marked line
$a =& new A(10);
$a->createRef();

$a->echoValue();
$a->b->echoValue();
$a->c->echoValue();

$a->value = 11;

$a->echoValue();
$a->b->echoValue(); // *
$a->c->echoValue();

/*
output:
class A: 10
class B: 10
class B: 10
class A: 11
class B: 11
class B: 11
*/
```

Chapter 15. References Explained