Error Handling and Debugging

7

If you're working through this book sequentially (which would be for the best), the next subject to learn is how to use PHP and MySQL together. However, that process will undoubtedly generate errors, errors that can be tricky to debug. So before moving on to new concepts, these next few pages address the bane of the programmer: errors. As you gain experience, you'll make fewer errors and pick up your own debugging methods, but there are plenty of tools and techniques the beginner can use to help ease the learning process.

This chapter has three main threads. One focus is on learning about the various kinds of errors that can occur when developing dynamic Web sites and what their likely causes are. Second, a multitude of debugging techniques are taught, in a step-by-step format. Finally, you'll see different techniques for handling the errors that occur in the most graceful manner possible.

Before reading on, a word regarding errors: they happen to the best of us. Even the author of this here book sees more than enough errors in his Web development duties (but rest assured that the code in this book should be bug-free). Thinking that you'll get to a skill level where errors never occur is a fool's dream, but there are techniques for minimizing errors, and knowing how to quickly catch, handle, and fix errors is a major skill in its own right. So try not to become frustrated as you make errors; instead, bask in the knowledge that you're becoming a better debugger!

Error Types and Basic Debugging

When developing Web applications with PHP and MySQL, you end up with potential bugs in one of four or more technologies. You could have HTML issues, PHP problems, SQL errors, or MySQL mistakes. To be able to stop the bugs, you must first find the crack they're sneaking in through.

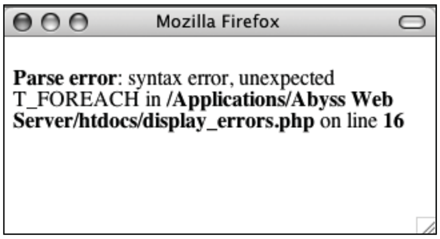
HTML problems are often the least disruptive and the easiest to catch. You normally know there's a problem when your layout is all messed up. Some steps for catching and fixing these, as well as general debugging hints, are discussed in the next section.

PHP errors are the ones you'll see most often, as this language will be at the heart of your applications. PHP errors fall into three general areas:

- Syntactical
- Run time
- Logical

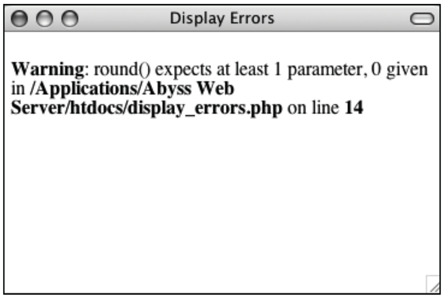
Syntactical errors are the most common and the easiest to fix. You'll see them if you merely omit a semicolon. Such errors stop the script from executing, and if *display_errors* is on in your PHP configuration, PHP will show an error, including the line PHP thinks it's on (**Figure 7.1**). If *display_errors* is off, you'll see a blank page. (You'll learn more about *display_errors* later in this chapter.)

Run-time errors include those things that don't stop a PHP script from executing (like parse errors do) but do stop the script from doing everything it was supposed to do. Examples include calling a function using the wrong number or types of parameters. With these errors, PHP will normally display a message (**Figure 7.2**) indicating the exact problem (again, assuming that *display_errors* is on).



Parse error: syntax error, unexpected T_FOREACH in **/Applications/Abyss Web Server/htdocs/display_errors.php** on line **16**

Figure 7.1 Parse errors—which you've probably seen many times over by now—are the most common sort of PHP error, particularly for beginning programmers.



Warning: round() expects at least 1 parameter, 0 given in **/Applications/Abyss Web Server/htdocs/display_errors.php** on line **14**

Figure 7.2 Misusing a function (calling it with improper parameters) will create errors during the execution of the script.

The final category of error—logical—is actually the worst, because PHP won't necessarily report it to you. These are out-and-out bugs: problems that aren't obvious and don't stop the execution of a script. Tricks for solving all of these PHP errors will be demonstrated in just a few pages.

SQL errors are normally a matter of syntax, and they'll be reported when you try to run the query on MySQL. For example, I've done this many times (**Figure 7.3**):

```
DELETE * FROM tablename
```

The syntax is just wrong, a confusion with the `SELECT` syntax (`SELECT * FROM` *`tablename`*). The right syntax is

```
DELETE FROM tablename
```

Again, MySQL will raise a red flag when you have SQL errors, so these aren't that difficult to find and fix. With dynamic Web sites, the catch is that you don't always have static queries, but rather ones dynamically generated by PHP. In such cases, if there's a syntax problem, the issue is probably in your PHP code.

Besides reporting on SQL errors, MySQL has its own errors to consider. An inability to access the database is a common one and a showstopper at that (**Figure 7.4**). You'll also see errors when you misuse a MySQL function or ambiguously refer to a column in a join. Again, MySQL will report any such error in specific detail. Keep in mind that when a query doesn't return the records or otherwise have the result you expect, that's not a MySQL or SQL error, but rather a logical one. Toward the end of this chapter you'll see how to solve SQL and MySQL problems.

But as you have to walk before you can run, the next section covers the fundamentals of debugging dynamic Web sites, starting with the basic checks you should make and how to fix HTML problems.

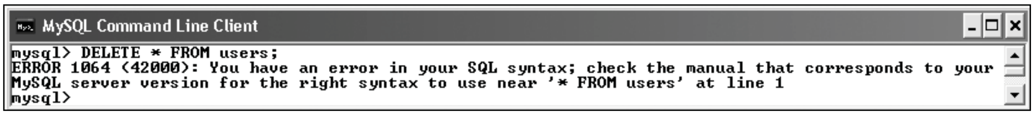
Basic debugging steps

This first sequence of steps may seem obvious, but when it comes to debugging, missing one of these steps leads to an unproductive and extremely frustrating debugging experience. And while I'm at it, I should mention that the best piece of general debugging advice is this:

When you get frustrated, step away from the computer!

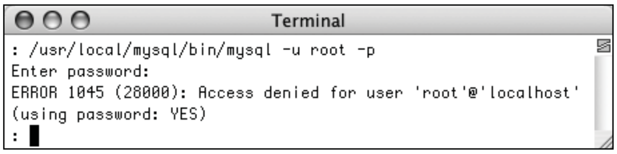
I have solved almost all of the most perplexing issues I've come across by taking a break, clearing my head, and coming back to the

continues on next page



```
mysql> DELETE * FROM users;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your
MySQL server version for the right syntax to use near '* FROM users' at line 1
mysql>
```

Figure 7.3 MySQL will report any errors found in the syntax of an SQL command.



```
: /usr/local/mysql/bin/mysql -u root -p
Enter password:
ERROR 1045 (28000): Access denied for user 'root'@'localhost'
(using password: YES)
:
```

Figure 7.4 An inability to connect to a MySQL server or a specific database is a common MySQL error.

code with fresh eyes. Readers in the book's supporting forum (`www.DMCInsights.com/phorum/`) have frequently found this to be true as well. Trying to forge ahead when you're frustrated tends to make things worse.

To begin debugging any problem:

- Make sure that you are running the right page.

  It's altogether too common that you try to fix a problem and no matter what you do, it never goes away. The reason: you've actually been editing a different page than you thought.

- Make sure that you have saved your latest changes.

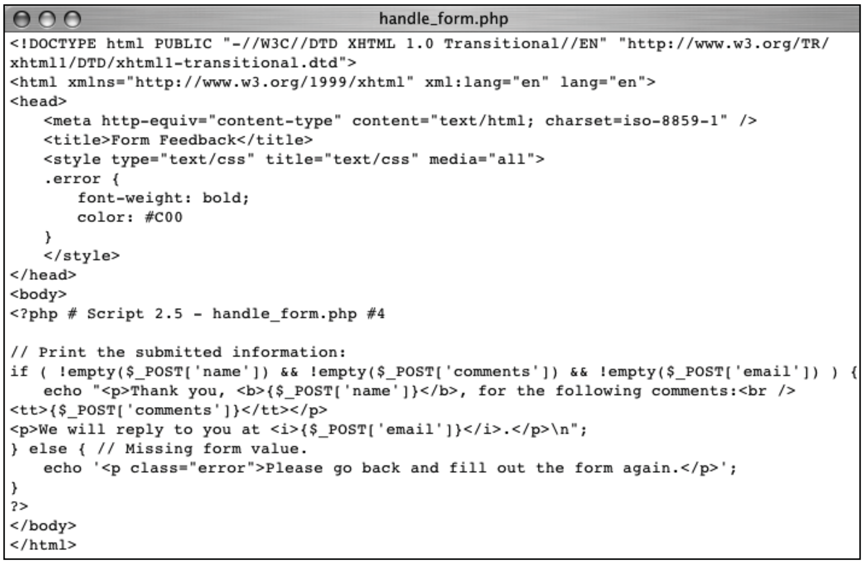
  An unsaved document will continue to have the same problems it had before you edited it (because the edits haven't been enacted).

- Make sure that you run all PHP pages through the URL.

  Because PHP works through a Web server (Apache, IIS, etc.), running any PHP code requires that you access the page through a URL (`http://www.example.com/page.php` or `http://localhost/page.php`). If you double-click a PHP page to open it in a browser (or use the browser's File > Open option), you'll see the PHP code, not the executed result. This also occurs if you load an HTML page without going through a URL (which will work on its own) but then submit the form to a PHP page (**Figure 7.5**).

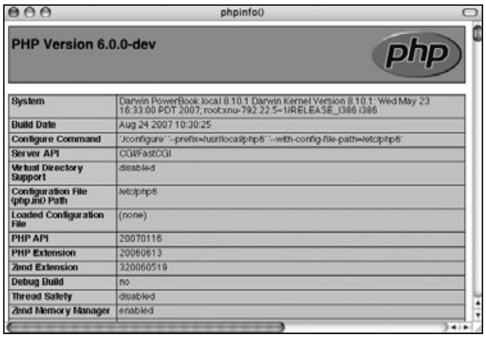
- Know what versions of PHP and MySQL you are running.

  Some problems are specific to a certain version of PHP or MySQL. For example, some functions are added in later versions of PHP, and MySQL added significant new features in versions 4, 4.1, and 5. Run a `phpinfo()` script (**Figure 7.6**, see Appendix A, "Installation," for a script example) and open a mysql client session



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
    <title>Form Feedback</title>
    <style type="text/css" title="text/css" media="all">
    .error {
        font-weight: bold;
        color: #C00
    }
    </style>
</head>
<body>
<?php # Script 2.5 - handle_form.php #4

// Print the submitted information:
if ( !empty($_POST['name']) && !empty($_POST['comments']) && !empty($_POST['email']) ) {
    echo "<p>Thank you, <b>{$_POST['name']}</b>, for the following comments:<br />
<tt>{$_POST['comments']}</tt></p>
<p>We will reply to you at <i>{$_POST['email']}</i>.</p>\n";
} else { // Missing form value.
    echo '<p class="error">Please go back and fill out the form again.</p>';
}
?>
</body>
</html>
```

Figure 7.5 PHP code will only be executed if run through a URL. This means that forms that submit to a PHP page must also be loaded through `http://`.

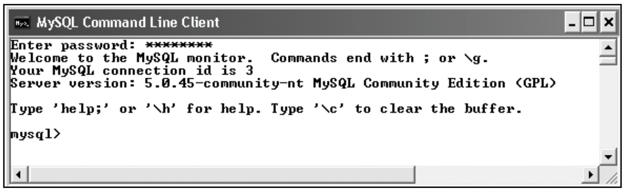


phpinfo()

PHP Version 6.0.0-dev

System	Darwin PowerBook.local 8.10.1 Darwin Kernel Version 8.10.1: Wed May 23 16:33:00 PDT 2007; root:xnu-792.22.5~1/RELEASE_I386 i386
Build Date	Aug 24 2007 10:30:25
Configure Command	'./configure' '--prefix=/usr/local/php6' '--with-config-file-path=/etc/php6'
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php6
Loaded Configuration File	(none)
PHP API	20070116
PHP Extension	20060613
Zend Extension	320060519
Debug Build	no
Thread Safety	disabled
Zend Memory Manager	enabled

Figure 7.6 A `phpinfo()` script is one of your best tools for debugging, informing you of the PHP version and how it's configured.



MySQL Command Line Client

```
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.0.45-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Figure 7.7 When you connect to a MySQL server, it should let you know the version number.

(**Figure 7.7**) to determine this information. phpMyAdmin will often report on the versions involved as well (but don't confuse the version of phpMyAdmin, which will likely be 2.*something* with the versions of PHP or MySQL).

I consider the versions being used to be such an important, fundamental piece of information that I won't normally assist people looking for help until they provide this information!

continues on next page

Book Errors

If you've followed an example in this book and something's not working right, what should you do?

1. Double-check your code or steps against those in the book.
2. Use the index at the back of the book to see if I reference a script or function in an earlier page (you may have missed an important usage rule or tip).
3. View the PHP manual for a specific function to see if it's available in your version of PHP and to verify how the function is used.
4. Check out the book's errata page (through the supporting Web site, `www.DMCInsights.com/phpmysql3/`) to see if an error in the code does exist and has been reported. Don't post your particular problem there yet, though!
5. Triple-check your code and use all the debugging techniques outlined in this chapter.
6. Search the book's supporting forum to see if others have had this problem and if a solution has already been determined.
7. If all else fails, use the book's supporting forum to ask for assistance. When you do, make sure you include all the pertinent information (version of PHP, version of MySQL, the debugging steps you took and what the results were, etc.).

- Know what Web server you are running. Similarly, some problems and features are unique to your Web serving application—Apache, IIS, or Abyss. You should know which one you are using, and which version, from when you installed the application.
- Try executing pages in a different Web browser.

  Every Web developer should have and use at least two Web browsers. If you test your pages in different ones, you'll see if the problem has to do with your script or a particular browser.
- If possible, try executing the page using a different Web server.

  PHP and MySQL errors sometimes stem from particular configurations and versions on one server. If something works on one server but not another, then you'll know that the script isn't inherently at fault. From there it's a matter of using `phpinfo()` scripts to see what server settings may be different.

✔ Tips

- If taking a break is one thing you should do when you become frustrated, here's what you *shouldn't* do: send off one or multiple panicky and persnickety emails to a writer, to a newsgroup or mailing list, or to anyone else. When it comes to asking for free help from strangers, patience and pleasantries garner much better and faster results.
- For that matter, I would highly advise against randomly guessing at solutions. I've seen far too many people only complicate matters further by taking stabs at solutions, without a full understanding of what the attempted changes should or should not do.
- There's another different realm of errors that you could classify as *usage* errors: what goes wrong when the site's user doesn't do what you thought they would. These are very difficult to find on your own because it's hard for the programmer to use an application in a way other than she intended. As a golden rule, write your code so that it doesn't break even if the user doesn't do anything right!

Debugging HTML

Debugging HTML is relatively easy. The source code is very accessible, most problems are overt, and attempts at fixing the HTML don't normally make things worse (as can happen with PHP). Still, there are some basic steps you should follow to find and fix an HTML problem.

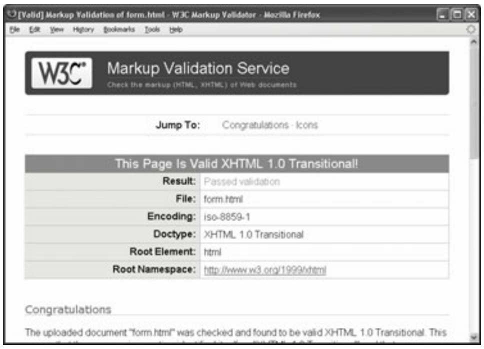
To debug an HTML error:

- Check the source code.

  If you have an HTML problem, you'll almost always need to check the source code of the page to find it. How you view the source code depends upon the browser being used, but normally it's a matter of using something like View > Page Source.
- Use a validation tool (**Figure 7.8**).

  Validation tools, like the one at `http://validator.w3.org`, are great for finding mismatched tags, broken tables, and other problems.
- Add borders to your tables.

  Frequently layouts are messed up because tables are incomplete. To confirm this, add a prominent border to your table to make it obvious where the different columns and rows are.



[Valid] Markup Validation of form.html - W3C Markup Validator - Mozilla Firefox

Markup Validation Service
Check the markup (HTML, XHTML) of Web documents

Jump To: Congratulations · Icons

This Page Is Valid XHTML 1.0 Transitional!

Result:	Passed validation
File:	form.html
Encoding:	iso-8859-1
Doctype:	XHTML 1.0 Transitional
Root Element:	html
Root Namespace:	http://www.w3.org/1999/xhtml

Congratulations

The uploaded document "form.html" was checked and found to be valid XHTML 1.0 Transitional. This

Figure 7.8 Validation tools like the one provided by the W3C (World Wide Web Consortium) are good for finding problems and making sure your HTML conforms to standards.

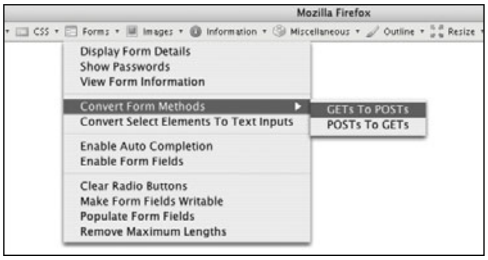


Figure 7.9 Firefox's Web Developer widget provides quick access to lots of useful tools.

✔ Tip

■ The first step toward fixing any kind of problem is understanding what's causing it. Remember the role each technology—HTML, PHP, SQL, and MySQL—plays as you debug. If your page doesn't look right, that's an HTML problem. If your HTML is dynamically generated by PHP, it's still an HTML problem but you'll need to work with the PHP code to make it right.

◆ Use Firefox or Opera.

I'm not trying to start a discussion on which is the best Web browser, and as Internet Explorer is the most used one, you'll need to eventually test using it, but I personally find that Firefox (available for free from `www.mozilla.com`) and Opera (available for free from `www.opera.com`) are the best Web browsers for Web developers. They offer reliability and debugging features not available in other browsers. If you want to stick with IE or Safari for your day-to-day browsing, that's up to you, but when doing Web development, start with either Firefox or Opera.

◆ Use Firefox's add-on widgets (**Figure 7.9**).

Besides being just a great Web browser, the very popular Firefox browser has a ton of features that the Web developer will appreciate. Furthermore, you can expand Firefox's functionality by installing any of the free widgets that are available. The Web Developer widget in particular provides quick access to great tools, such as showing a table's borders, revealing the CSS, validating a page, and more. I also frequently use these add-ons: DOM Inspector, Firebug, and HTML Validator, among others.

◆ Test the page in another browser.

PHP code is generally browser-independent, meaning you'll get consistent results regardless of the client. Not so with HTML. Sometimes a particular browser has a quirk that affects the rendered page. Running the same page in another browser is the easiest way to know if it's an HTML problem or a browser quirk.

Displaying PHP Errors

PHP provides remarkably useful and descriptive error messages when things go awry. Unfortunately, PHP doesn't show these errors when running using its default configuration. This policy makes sense for live servers, where you don't want the end users seeing PHP-specific error messages, but it also makes everything that much more confusing for the beginning PHP developer. To be able to see PHP's errors, you must turn on the *display_errors* directive, either in an individual script or for the PHP configuration as a whole.

To turn on *display_errors* in a script, use the `ini_set()` function. As its arguments, this function takes a directive name and what setting that directive should have:

```
ini_set('display_errors', 1);
```

Including this line in a script will turn on *display_errors* for that script. The only downside is that if your script has a syntax error that prevents it from running at all, then you'll still see a blank page. To have PHP display errors for the entire server, you'll need to edit its configuration, as is discussed in the "Configuring PHP" section of Appendix A.

To turn on display_errors:

1. Create a new PHP document in your text editor or IDE (**Script 7.1**).

   ```
   <!DOCTYPE html PUBLIC "-//W3C//DTD
   → XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/
   → xhtml1-transitional.dtd">
   <html xmlns="http://www.w3.org/1999/
   → xhtml" xml:lang="en" lang="en">
   <head>
   ```

Script 7.1 The `ini_set()` function can be used to tell a PHP script to reveal any errors that might occur.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
  <title>Display Errors</title>
</head>
<body>
<h2>Testing Display Errors</h2>
<?php # Script 7.1 - display_errors.php

// Show errors:
ini_set('display_errors', 1);

// Create errors:
foreach ($var as $v) {}
$result = 1/0;

?>
</body>
</html>
```

```
    <meta http-equiv="content-type"
    → content="text/html; charset=
    → iso-8859-1" />
    <title>Display Errors</title>
</head>
<body>
<?php # Script 7.1 - display_
→ errors.php
```

2. After the initial PHP tags, add

   ```
   ini_set('display_errors', 1);
   ```

   From this point in this script forward, any errors that occur will be displayed.

3. Create some errors.

   ```
   foreach ($var as $v) {}
   $result = 1/0;
   ```

   To test the *display_errors* setting, the script needs to have an error. This first line doesn't even try to do anything, but it's guaranteed to cause an error. There are actually two issues here: first, there's a reference to a variable (`$var`) that doesn't exist; second, a non-array (`$var`) is being used as an array in the `foreach` loop.

   The second line is a classic division by zero, which is not allowed in programming languages or in math.

4. Complete the page.

   ```
   ?>
   </body>
   </html>
   ```

5. Save the file as `display_errors.php`, place it in your Web directory, and test it in your Web browser (**Figure 7.10**).

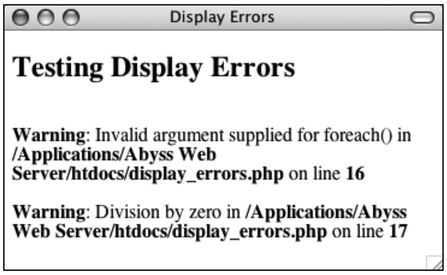
6. If you want, change the first line of PHP code to read

   ```
   ini_set('display_errors', 0);
   ```

   and then save and retest the script (**Figure 7.11**).

✔ Tips

- There are limits as to what PHP settings the `ini_set()` function can be used to adjust. See the PHP manual for specifics as to what can and cannot be changed using it.
- As a reminder, changing the *display_errors* setting in a script only works so long as that script runs (i.e., it cannot have any parse errors). To be able to *always* see any errors that occur, you'll need to enable *display_errors* in PHP's configuration file (again, see the appendix).

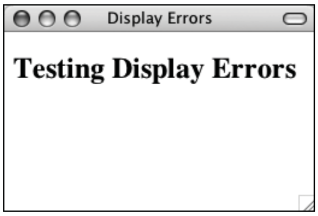


Testing Display Errors

Warning: Invalid argument supplied for foreach() in **/Applications/Abyss Web Server/htdocs/display_errors.php** on line **16**

Warning: Division by zero in **/Applications/Abyss Web Server/htdocs/display_errors.php** on line **17**

Figure 7.10 With *display_errors* turned on (for this script), the page reports the errors when they occur.



Testing Display Errors

Figure 7.11 With *display_errors* turned off (for this page), the same errors (Script 7.1 and Figure 7.10) are no longer reported. Unfortunately, they still exist.

Adjusting Error Reporting in PHP

Once you have PHP set to display the errors that occur, you might want to adjust the level of error reporting. Your PHP installation as a whole, or individual scripts, can be set to report or ignore different types of errors. **Table 7.1** lists most of the levels, but they can generally be one of these three kinds:

- *Notices*, which do not stop the execution of a script and may not necessarily be a problem.
- *Warnings*, which indicate a problem but don't stop a script's execution.
- *Errors*, which stop a script from continuing (including the ever-common parse error, which prevent scripts from running at all).

As a rule of thumb, you'll want PHP to report on any kind of error while you're developing a site but report no specific errors once the site goes live. For security and aesthetic purposes, it's generally unwise for a public user to see PHP's detailed error messages. Frequently, error messages—particularly those dealing with the database—will reveal

Suppressing Errors with @

Individual errors can be suppressed in PHP using the @ operator. For example, if you don't want PHP to report if it couldn't include a file, you would code

```
@include ('config.inc.php');
```

Or if you don't want to see a "division by zero" error:

```
$x = 8;
$y = 0;
$num = @($x/$y);
```

The @ symbol will work only on expressions, like function calls or mathematical operations. You cannot use @ before conditionals, loops, function definitions, and so forth.

As a rule of thumb, I recommend that @ be used on functions whose execution, should they fail, will not affect the functionality of the script as a whole. Or you can suppress PHP's errors when you will handle them more gracefully yourself (a topic discussed later in this chapter).

Table 7.1 PHP's error-reporting settings, to be used with the `error_reporting()` function or in the `php.ini` file. Note that `E_ALL`'s number value was different in earlier versions of PHP and did not include `E_STRICT` (it does in PHP 6).

Error-Reporting Levels

Number	Constant	Report On
1	E_ERROR	Fatal run-time errors (that stop execution of the script)
2	E_WARNING	Run-time warnings (non-fatal errors)
4	E_PARSE	Parse errors
8	E_NOTICE	Notices (things that could or could not be a problem)
256	E_USER_ERROR	User-generated error messages, generated by the `trigger_error()` function
512	E_USER_WARNING	User-generated warnings, generated by the `trigger_error()` function
1024	E_USER_NOTICE	User-generated notices, generated by the `trigger_error()` function
2048	E_STRICT	Recommendations for compatibility and interoperability
8191	E_ALL	All errors, warnings, and recommendations

Script 7.2 This script will demonstrate how error reporting can be manipulated in PHP.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/
    xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content=
  "text/html; charset=iso-8859-1" />
  <title>Report Errors</title>
</head>
<body>
<h2>Testing Error Reporting</h2>
<?php # Script 7.2 - report_errors.php

// Show errors:
ini_set('display_errors', 1);

// Adjust error reporting:
error_reporting(E_ALL);

// Create errors:
foreach ($var as $v) {}
$result = 1/0;

?>
</body>
</html>
```

certain behind-the-scenes aspects of your Web application that are best not shown. While you hope all of these will be worked out during the development stages, that may not be the case.

You can universally adjust the level of error reporting following the instructions in Appendix A. Or you can adjust this behavior on a script-by-script basis using the `error_reporting()` function. This function is used to establish what type of errors PHP should report on within a specific page. The function takes either a number or a constant, using the values in Table 7.1 (the PHP manual lists a few others, related to the core of PHP itself).

```
error_reporting(0); // Show no errors.
```

A setting of 0 turns error reporting off entirely (errors will still occur; you just won't see them anymore). Conversely, `error_reporting (E_ALL)` will tell PHP to report on every error that occurs. The numbers can be added up to customize the level of error reporting, or you could use the bitwise operators—`|` (or), `~` (not), `&` (and)—with the constants. With this following setting any non-notice error will be shown:

```
error_reporting (E_ALL & ~E_NOTICE);
```

To adjust error reporting:

1. Open `display_errors.php` (Script 7.1) in your text editor or IDE.

   To play around with error reporting levels, use `display_errors.php` as an example.

2. After adjust the *display_errors* setting, add (**Script 7.2**)

   ```
   error_reporting (E_ALL);
   ```

   For development purposes, have PHP notify you of all errors, notices, warnings, and recommendations. This line will

continues on next page

accomplish that. In short, PHP will let you know about anything that is, or may be, a problem.

Because `E_ALL` is a constant, it is not enclosed in quotation marks.

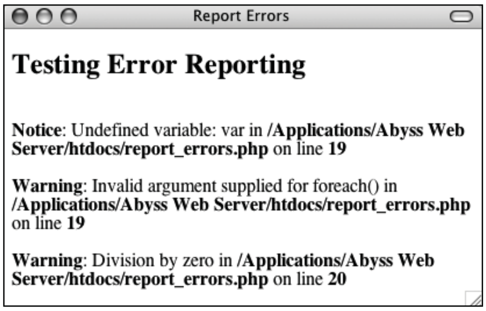
3. Save the file as `report_errors.php`, place it in your Web directory, and run it in your Web browser (**Figure 7.12**).

   I also altered the page's title and the heading, but both are immaterial to the point of this exercise.

4. Change the level of error reporting to something different and retest (**Figures 7.13** and **7.14**).

✔ Tips

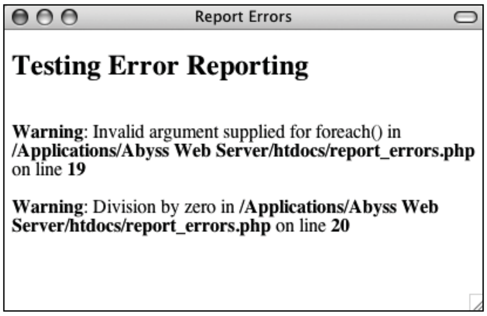
- Because you'll often want to adjust the *display_errors* and *error_reporting* for every page in a Web site, you might want to place those lines of code in a separate PHP file that can then be included by other PHP scripts.
- In case you are curious, the scripts in this book were all written with PHP's error reporting on the highest level (with the intention of catching every possible problem).


Report Errors

Testing Error Reporting

```
Notice: Undefined variable: var in /Applications/Abyss Web Server/htdocs/report_errors.php on line 19

Warning: Invalid argument supplied for foreach() in /Applications/Abyss Web Server/htdocs/report_errors.php on line 19

Warning: Division by zero in /Applications/Abyss Web Server/htdocs/report_errors.php on line 20
```

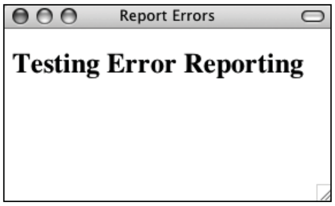
Figure 7.12 On the highest level of error reporting, PHP has two warnings and one notice for this page (Script 7.2).


Report Errors

Testing Error Reporting

```
Warning: Invalid argument supplied for foreach() in /Applications/Abyss Web Server/htdocs/report_errors.php on line 19

Warning: Division by zero in /Applications/Abyss Web Server/htdocs/report_errors.php on line 20
```

Figure 7.13 The same page (Script 7.2) after disabling the reporting of notices.


Report Errors

Testing Error Reporting

Figure 7.14 The same page again (Script 7.2) with error reporting turned off (set to 0). The result is the same as if *display_errors* was disabled. Of course, the errors still occur; they're just not being reported.

Creating Custom Error Handlers

Another option for error management with your sites is to alter how PHP handles errors. By default, if *display_errors* is enabled and an error is caught (that falls under the level of error reporting), PHP will print the error, in a somewhat simplistic form, within some minimal HTML tags (**Figure 7.15**).

You can override how errors are handled by creating your own function that will be called when errors occur. For example,

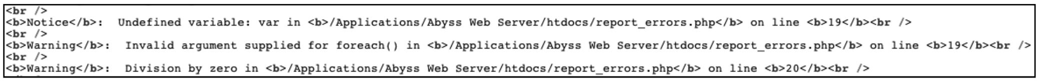
```
function report_errors (arguments) {
   // Do whatever here.
}
set_error_handler ('report_errors');
```

The `set_error_handler()` function is used to name the function to be called when an error occurs. The handling function (*report_errors*, in this case) will, at that time, receive several values that can be used in any possible manner.

This function can be written to take up to five arguments. In order, these arguments are: an error number (corresponding to Table 7.1), a textual error message, the name of the file where the error was found, the specific line number on which it occurred, and the variables that existed at the time of the error. Defining a function that accepts all these arguments might look like

```
function report_errors ($num, $msg,
$file, $line, $vars) {…
```

To make use of this concept, the `report_errors.php` file (Script 7.2) will be rewritten one last time.



```
<br />
<b>Notice</b>:  Undefined variable: var in <b>/Applications/Abyss Web Server/htdocs/report_errors.php</b> on line <b>19</b><br />
<br />
<b>Warning</b>:  Invalid argument supplied for foreach() in <b>/Applications/Abyss Web Server/htdocs/report_errors.php</b> on line <b>19</b><br />
<br />
<b>Warning</b>:  Division by zero in <b>/Applications/Abyss Web Server/htdocs/report_errors.php</b> on line <b>20</b><br />
```

Figure 7.15 The HTML source code for the errors shown in Figure 7.12.

To create your own error handler:

1. Open `report_errors.php` (Script 7.2) in your text editor or IDE.

2. Remove the `ini_set()` and `error_reporting()` lines (**Script 7.3**).

   When you establish your own error handling function, the error reporting levels no longer have any meaning, so that line can be removed. Adjusting the *display_errors* setting is also meaningless, as the error handling function will control whether errors are displayed or not.

3. Before the script creates the errors, add

   ```
   define ('LIVE', FALSE);
   ```

   This constant will be a flag used to indicate whether or not the site is currently live. It's an important distinction, as how you handle errors and what you reveal in the browser should differ greatly when you're developing a site and when a site is live.

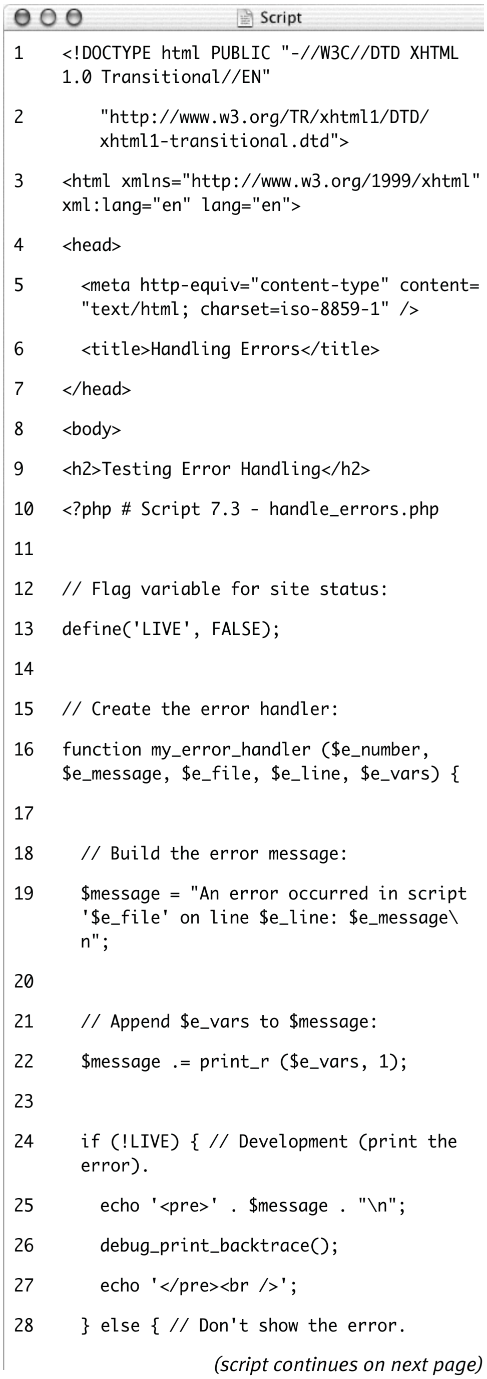
   This constant is being set outside of the function for two reasons. First, I want to treat the function as a black box that does what I need it to do without having to go in and tinker with it. Second, in many sites, there might be other settings (like the database connectivity information) that are also live versus development-specific. Conditionals could, therefore, also refer to this constant to adjust those settings.

4. Begin defining the error handling function.

   ```
   function my_error_handler ($e_number,
   → $e_message, $e_file, $e_line,
   → $e_vars) {
   ```

   The `my_error_handler()` function is set to receive the full five arguments that a custom error handler can.

Script 7.3 By defining your own error handling function, you can customize how errors are treated in your PHP scripts.



Script

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
  <title>Handling Errors</title>
</head>
<body>
<h2>Testing Error Handling</h2>
<?php # Script 7.3 - handle_errors.php

// Flag variable for site status:
define('LIVE', FALSE);

// Create the error handler:
function my_error_handler ($e_number, $e_message, $e_file, $e_line, $e_vars) {

  // Build the error message:
  $message = "An error occurred in script '$e_file' on line $e_line: $e_message\n";

  // Append $e_vars to $message:
  $message .= print_r ($e_vars, 1);

  if (!LIVE) { // Development (print the error).
    echo '<pre>' . $message . "\n";
    debug_print_backtrace();
    echo '</pre><br />';
  } else { // Don't show the error.
```

(script continues on next page)

Script 7.3 *continued*

```
        echo '<div class="error">A system
        error occurred. We apologize for
        the inconvenience.</div><br />';
    }

} // End of my_error_handler() definition.

// Use my error handler:
set_error_handler ('my_error_handler');

// Create errors:
foreach ($var as $v) {}
$result = 1/0;

?>
</body>
</html>
```

5. Create the error message using the received values.

   ```
   $message = "An error occurred in
   → script '$e_file' on line $e_line:
   → $e_message\n";
   ```

   The error message will begin by referencing the filename and number where the error occurred. Added to this is the actual error message. All of these values are passed to the function when it is called (when an error occurs).

6. Add any existing variables to the error message.

   ```
   $message .= print_r ($e_vars, 1);
   ```

   The `$e_vars` variable will receive all of the variables that exist, and their values, when the error happens. Because this might contain useful debugging information, it's added to the message.

   The `print_r()` function is normally used to print out a variable's structure and value; it is particularly useful with arrays. If you call the function with a second argument (*1* or `TRUE`), the result is returned instead of printed. So this line adds all of the variable information to `$message`.

7. Print a message that will vary, depending upon whether or not the site is live.

   ```
   if (!LIVE) {
       echo '<pre>' . $message . "\n";
       debug_print_backtrace();
       echo '</pre><br />';
   } else {
       echo '<div class="error">A
       → system error occurred. We
       → apologize for the
       → inconvenience.</div><br />';
   }
   ```

continues on next page

If the site is not live (if `LIVE` is false), which would be the case while the site is being developed, a detailed error message should be printed (**Figure 7.16**). For ease of viewing, the error message is printed within HTML `PRE` tags (which aren't XHMTL valid but are very helpful here). Furthermore, a useful debugging function, `debug_print_backtrace()`, is also called. This function returns a slew of information about what functions have been called, what files have been included, and so forth.

If the site is live, a simple mea culpa will be printed, letting the user know that an error occurred but not what the specific problem is (**Figure 7.17**). Under this situation, you could also use the `error_log()` function (see the sidebar) to have the detailed error message emailed or written to a log.

8. Complete the function and tell PHP to use it.

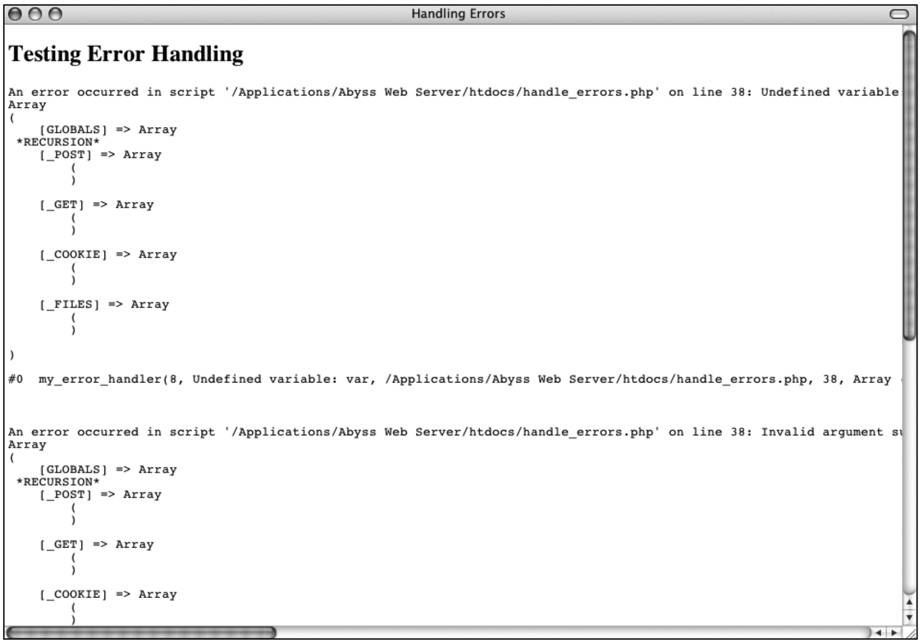
   ```
   }
   set_error_handler('my_error_handler'
   → );
   ```

   This second line is the important one, telling PHP to use the custom error handler instead of PHP's default handler.

9. Save the file as `handle_errors.php`, place it in your Web directory, and test it in your Web browser (Figure 7.16).

10. Change the value of `LIVE` to *TRUE*, save, and retest the script (Figure 7.17).

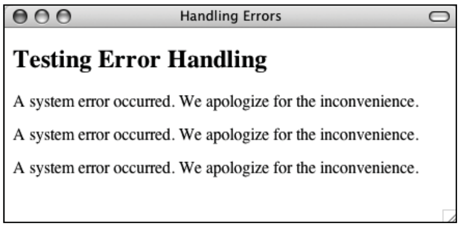
    To see how the error handler behaves with a live site, just change this one value.



Testing Error Handling

```
An error occurred in script '/Applications/Abyss Web Server/htdocs/handle_errors.php' on line 38: Undefined variable
Array
(
    [GLOBALS] => Array
 *RECURSION*
    [_POST] => Array
        (
        )

    [_GET] => Array
        (
        )

    [_COOKIE] => Array
        (
        )

    [_FILES] => Array
        (
        )

)
#0  my_error_handler(8, Undefined variable: var, /Applications/Abyss Web Server/htdocs/handle_errors.php, 38, Array


An error occurred in script '/Applications/Abyss Web Server/htdocs/handle_errors.php' on line 38: Invalid argument s
Array
(
    [GLOBALS] => Array
 *RECURSION*
    [_POST] => Array
        (
        )

    [_GET] => Array
        (
        )

    [_COOKIE] => Array
        (
        )
```

Figure 7.16 During the development phase, detailed error messages are printed in the Web browser. (In a more real-world script, with more code, the messages would be more useful.)


Handling Errors

Testing Error Handling

A system error occurred. We apologize for the inconvenience.

A system error occurred. We apologize for the inconvenience.

A system error occurred. We apologize for the inconvenience.

Figure 7.17 Once a site has gone live, more user-friendly (and less revealing) errors are printed. Here, one message is printed for each of the three errors in the script.

Logging PHP Errors

In Script 7.3, errors are handled by simply printing them out in detail or not. Another option is to log the errors: make a permanent note of them somehow. For this purpose, the `error_log()` function instructs PHP how to file an error. It's syntax is

```
error_log (message, type,
→ destination,
extra headers);
```

The *message* value should be the text of the logged error (i.e., `$message` in Script 7.3). The *type* dictates how the error is logged. The options are the numbers 0 through 3: use the computer's default logging method (0), send it in an email (1), send to a remote debugger (2), or write it to a text file (3).

The *destination* parameter can be either the name of a file (for log type 3) or an email address (for log type 1). The *extra headers* argument is used only when sending emails (log type 1). Both the destination and extra headers are optional.

✔ Tips

- If your PHP page uses special HTML formatting—like CSS tags to affect the layout and font treatment—add this information to your error reporting function.
- Obviously in a live site you'll probably need to do more than apologize for the inconvenience (particularly if the error significantly affects the page's functionality). Still, this example demonstrates how you can easily adjust error handling to suit the situation.
- If you don't want the error handling function to report on every notice, error, or warning, you could check the error number value (the first argument sent to the function). For example, to ignore notices when the site is live, you would change the main conditional to

```
if (!LIVE) {
    echo '<pre>' . $message . "\n";
    debug_print_backtrace();
    echo '</pre><br />';
} elseif ($e_number != E_NOTICE) {
    echo '<div class="error">A
    → system error occurred. We
    → apologize for the
    → inconvenience.</div><br />';
}
```

- You can invoke your error handling function using `trigger_error()`.

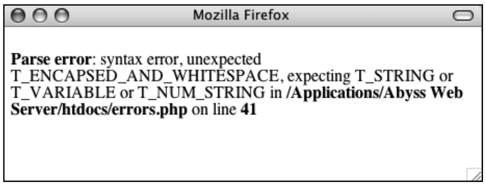
PHP Debugging Techniques

When it comes to debugging, what you'll best learn from experience are the causes of certain types of errors. Understanding the common causes will shorten the time it takes to fix errors. To expedite the learning process, **Table 7.2** lists the likely reasons for the most common PHP errors.

The first, and most common, type of error that you'll run across is syntactical and will prevent your scripts from executing. An error like this will result in messages like the one in **Figure 7.18**, which every PHP developer has seen too many times. To avoid making this sort of mistake when you program, be sure to:

- ◆ End every statement (but not language constructs like loops and conditionals) with a semicolon.
- ◆ Balance all quotation marks, parentheses, curly braces, and square brackets (each opening character must be closed).
- ◆ Be consistent with your quotation marks (single quotes can be closed only with single quotes and double quotes with double quotes).
- ◆ Escape, using the backslash, all single- and double-quotation marks within strings, as appropriate.

One thing you should also understand about syntactical errors is that just because the PHP error message says the error is occurring on line 12, that doesn't mean that the mistake is actually on that line. At the very least, it is not uncommon for there to be



```
Parse error: syntax error, unexpected
T_ENCAPSED_AND_WHITESPACE, expecting T_STRING or
T_VARIABLE or T_NUM_STRING in /Applications/Abyss Web
Server/htdocs/errors.php on line 41
```

Figure 7.18 The parse error prevents a script from running because of invalid PHP syntax. This one was caused by failing to enclose `$array['key']` within curly braces when printing its value.

Table 7.2 These are some of the most common errors you'll see in PHP, along with their most probable causes.

Common PHP Errors

Error	Likely Cause
Blank Page	HTML problem, or PHP error and *display_errors* or *error_reporting* is off.
Parse error	Missing semicolon; unbalanced curly braces, parentheses, or quotation marks; or use of an unescaped quotation mark in a string.
Empty variable value	Forgot the initial $, misspelled or miscapitalized the variable name, or inappropriate variable scope (with functions).
Undefined variable	Reference made to a variable before it is given a value or an empty variable value (see those potential causes).
Call to undefined function	Misspelled function name, PHP is not configured to use that function (like a MySQL function), or document that contains the function definition was not included.
Cannot redeclare function	Two definitions of your own function exist; check within included files.
Headers already sent	White space exists in the script before the PHP tags, data has already been printed, or a file has been included.

a difference between what PHP thinks is line 12 and what your text editor indicates is line 12. So while PHP's direction is useful in tracking down a problem, treat the line number referenced as more of a starting point than an absolute.

If PHP reports an error on the last line of your document, this is almost always because a mismatched parenthesis, curly brace, or quotation mark was not caught until that moment.

The second type of error you'll encounter results from misusing a function. This error occurs, for example, when a function is called without the proper arguments. This error is discovered by PHP when attempting to execute the code. In later chapters you'll probably see such errors when using the `header()` function, cookies, or sessions.

To fix errors, you'll need to do a little detective work to see what mistakes were made and where. For starters, though, always thoroughly read and trust the error message PHP offers. Although the referenced line number may not always be correct, a PHP error is very descriptive, normally helpful, and almost always 100 percent correct.

To debug your scripts:

- Turn on *display_errors*.

  Use the earlier steps to enable *display_errors* for a script, or, if possible, the entire server, as you develop your applications.

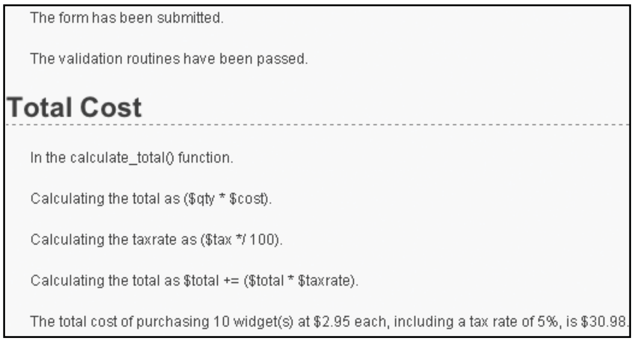
- Use comments.

  Just as you can use comments to document your scripts, you can also use them to rule out problematic lines. If PHP is giving you an error on line 12, then commenting out that line should get rid of the error. If not, then you know the error is elsewhere. Just be careful that you don't introduce more errors by improperly commenting out only a portion of a code block: the syntax of your scripts must be maintained.

- Use the `print()` and `echo()` functions.

  In more complicated scripts, I frequently use `echo()` statements to leave me notes as to what is happening as the script is executed (**Figure 7.19**). When a script has several steps, it may not be easy to know if the problem is occurring in step 2 or step 5. By using an `echo()` statement, you can narrow the problem down to the specific juncture.

continues on next page



```
The form has been submitted.

The validation routines have been passed.

Total Cost

In the calculate_total() function.

Calculating the total as ($qty * $cost).

Calculating the taxrate as ($tax */ 100).

Calculating the total as $total += ($total * $taxrate).

The total cost of purchasing 10 widget(s) at $2.95 each, including a tax rate of 5%, is $30.98.
```

Figure 7.19 More complex debugging can be accomplished by leaving yourself notes as to what the script is doing.

- Check what quotation marks are being used for printing variables.

  It's not uncommon for programmers to mistakenly use single quotation marks and then wonder why their variables are not printed properly. Remember that single quotation marks treat text literally and that you must use double quotation marks to print out the values of variables.

- Track variables (**Figure 7.20**).

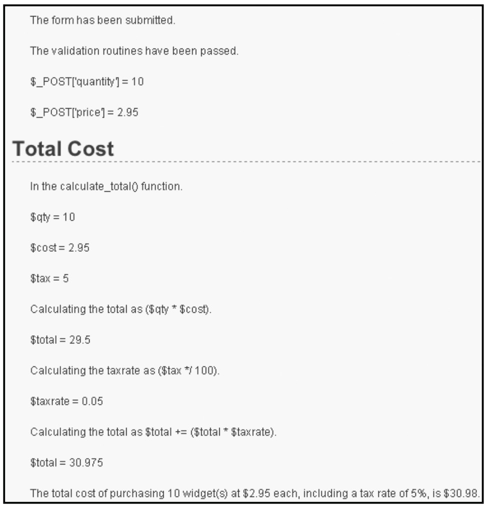
  It is pretty easy for a script not to work because you referred to the wrong variable or the right variable by the wrong name or because the variable does not have the value you would expect. To check for these possibilities, use the `print()` or `echo()` statements to print out the values of variables at important points in your scripts. This is simply a matter of

  ```
  echo "<p>\$var = $var</p>\n";
  ```

  The first dollar sign is escaped so that the variable's name is printed. The second reference of the variable will print its value.

- Print array values.

  For more complicated variable types (arrays and objects), the `print_r()` and `var_dump()` functions will print out their values without the need for loops. Both functions accomplish the same task, although `var_dump()` is more detailed in its reporting than `print_r()`.



```
The form has been submitted.

The validation routines have been passed.

$_POST['quantity'] = 10

$_POST['price'] = 2.95

Total Cost

In the calculate_total() function.

$qty = 10

$cost = 2.95

$tax = 5

Calculating the total as ($qty * $cost).

$total = 29.5

Calculating the taxrate as ($tax */ 100).

$taxrate = 0.05

Calculating the total as $total += ($total * $taxrate).

$total = 30.975

The total cost of purchasing 10 widget(s) at $2.95 each, including a tax rate of 5%, is $30.98.
```

Figure 7.20 Printing the names and values of variables is the easiest way to track them over the course of a script.

✔ Tips

- Many text editors include utilities to check for balanced parentheses, brackets, and quotation marks.
- If you cannot find the parse error in a complex script, begin by using the `/* */` comments to render the entire PHP code inert. Then continue to uncomment sections at a time (by moving the opening or closing comment characters) and rerun the script until you deduce what lines are causing the error. Watch how you comment out control structures, though, as the curly braces must continue to be matched in order to avoid parse errors. For example:

```
if (condition) {
    /* Start comment.
    Inert code.
    End comment. */
}
```

- To make the results of `print_r()` more readable in the Web browser, wrap it within HTML `<pre>` (preformatted) tags. This one line is my absolute favorite debugging tool:

```
echo '<pre>' . print_r ($var, 1) .
→ '</pre>';
```

Using die() and exit()

Two functions that are often used with error management are `die()` and `exit()`, (they're technically language constructs, not functions, but who cares?). When a `die()` or `exit()` is called in your script, the entire script is terminated. Both are useful for stopping a script from continuing should something important—like establishing a database connection—fail to happen. You can also pass `die()` and `exit()` a string that will be printed out in the browser.

You'll commonly see `die()` or `exit()` used in an `OR` conditional. For example:

```
include('config.inc.php') OR die
→ ('Could not open the file. ');
```

With a line like that, if PHP could not include the configuration file, the die() statement will be executed and the "Could not open the file." message will be printed. You'll see variations on this throughout this book and in the PHP manual, as it's a quick (but potentially excessive) way to handle errors without using a custom error handler.

SQL and MySQL Debugging Techniques

The most common SQL errors are caused by the following issues:

- Unbalanced use of quotation marks or parentheses
- Unescaped apostrophes in column values
- Misspelling a column name, table name, or function
- Ambiguously referring to a column in a join
- Placing a query's clauses (`WHERE`, `GROUP BY`, `ORDER BY`, `LIMIT`) in the wrong order

Furthermore, when using MySQL you can also run across the following:

- Unpredictable or inappropriate query results
- Inability to access the database

Since you'll be running the queries for your dynamic Web sites from PHP, you need a methodology for debugging SQL and MySQL errors within that context (PHP will not report a problem with your SQL).

Debugging SQL problems

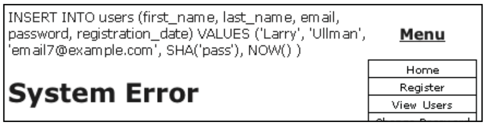
To decide if you are experiencing a MySQL (or SQL) problem rather than a PHP one, you need a system for finding and fixing the issue. Fortunately, the steps you should take to debug MySQL and SQL problems are easy to define and should be followed without thinking. If you ever have any MySQL or SQL errors to debug, just abide by this sequence of steps.

To hammer the point home, this next sequence of steps is probably the most useful debugging technique in this chapter and the entire book. You'll likely need to follow these steps in any PHP-MySQL Web application when you're not getting the results you expected.

To debug your SQL queries:

1. Print out any applicable queries in your PHP script (**Figure 7.21**).

   As you'll see in the next chapter, SQL queries will often be assigned to a variable, particularly when you use PHP to dynamically write them. Using the code `echo $query` (or whatever the query variable is called) in your PHP scripts, you can send to the browser the exact query being run. Sometimes this step alone will help you see what the real problem is.



```
INSERT INTO users (first_name, last_name, email,
password, registration_date) VALUES ('Larry', 'Ullman',
'email7@example.com', SHA('pass'), NOW() )
```

System Error

Figure 7.21 Knowing exactly what query a PHP script is attempting to execute is the most useful first step for solving SQL and MySQL problems.

2. Run the query in the mysql client or other tool (**Figure 7.22**).

   The most foolproof method of debugging an SQL or MySQL problem is to run the query used in your PHP scripts through an independent application: the mysql client, phpMyAdmin, or the like. Doing so will give you the same result as the original PHP script receives but without the overhead and hassle.

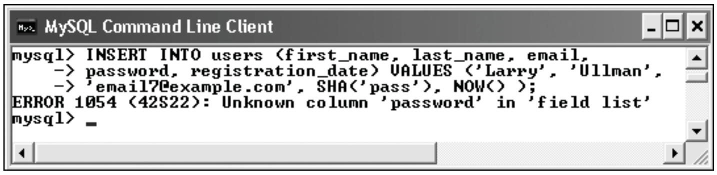
   If the independent application returns the expected result but you are still not getting the proper behavior in your PHP script, then you will know that the problem lies within the script itself, not your SQL or MySQL database.

3. If the problem still isn't evident, rewrite the query in its most basic form, and then keep adding dimensions back in until you discover which clause is causing the problem.

   Sometimes it's difficult to debug a query because there's too much going on. Like commenting out most of a PHP script, taking a query down to its bare minimum structure and slowly building it back up can be the easiest way to debug complex SQL commands.

✔ Tips

- Another common MySQL problem is trying to run queries or connect using the mysql client when the MySQL server isn't even running. Be sure that MySQL is available for querying!
- As an alternative to printing out the query to the browser, you could print it out as an HTML comment (viewable only in the HTML source), using
  `echo "<!-- $query -->";`



```
mysql> INSERT INTO users (first_name, last_name, email,
    -> password, registration_date) VALUES ('Larry', 'Ullman',
    -> 'email7@example.com', SHA('pass'), NOW() );
ERROR 1054 (42S22): Unknown column 'password' in 'field list'
mysql> _
```

Figure 7.22 To understand what result a PHP script is receiving, run the same query through a separate interface. In this case the problem is the reference to the *password* column, when the table's column is actually called just *pass*.

Debugging access problems

Access denied error messages are the most common problem beginning developers encounter when using PHP to interact with MySQL. These are among the common solutions:

- Reload MySQL after altering the privileges so that the changes take effect. Either use the mysqladmin tool or run `FLUSH PRIVILEGES` in the mysql client. You must be logged in as a user with the appropriate permissions to do this (see Appendix A for more).
- Double-check the password used. The error message *Access denied for user: 'user@localhost' (Using password: YES)* frequently indicates that the password is wrong or mistyped. (This is not always the cause but is the first thing to check.)
- The error message *Can't connect to...* (error number 2002) indicates that MySQL either is not running or is not running on the socket or TCP/IP port tried by the client.

✔ Tips

- MySQL keeps its own error logs, which are very useful in solving MySQL problems (like why MySQL won't even start). MySQL's error log will be located in the data directory and titled ***hostname*.err**.
- The MySQL manual is very detailed, containing SQL examples, function references, and the meanings of error codes. Make the manual your friend and turn to it when confusing errors pop up.