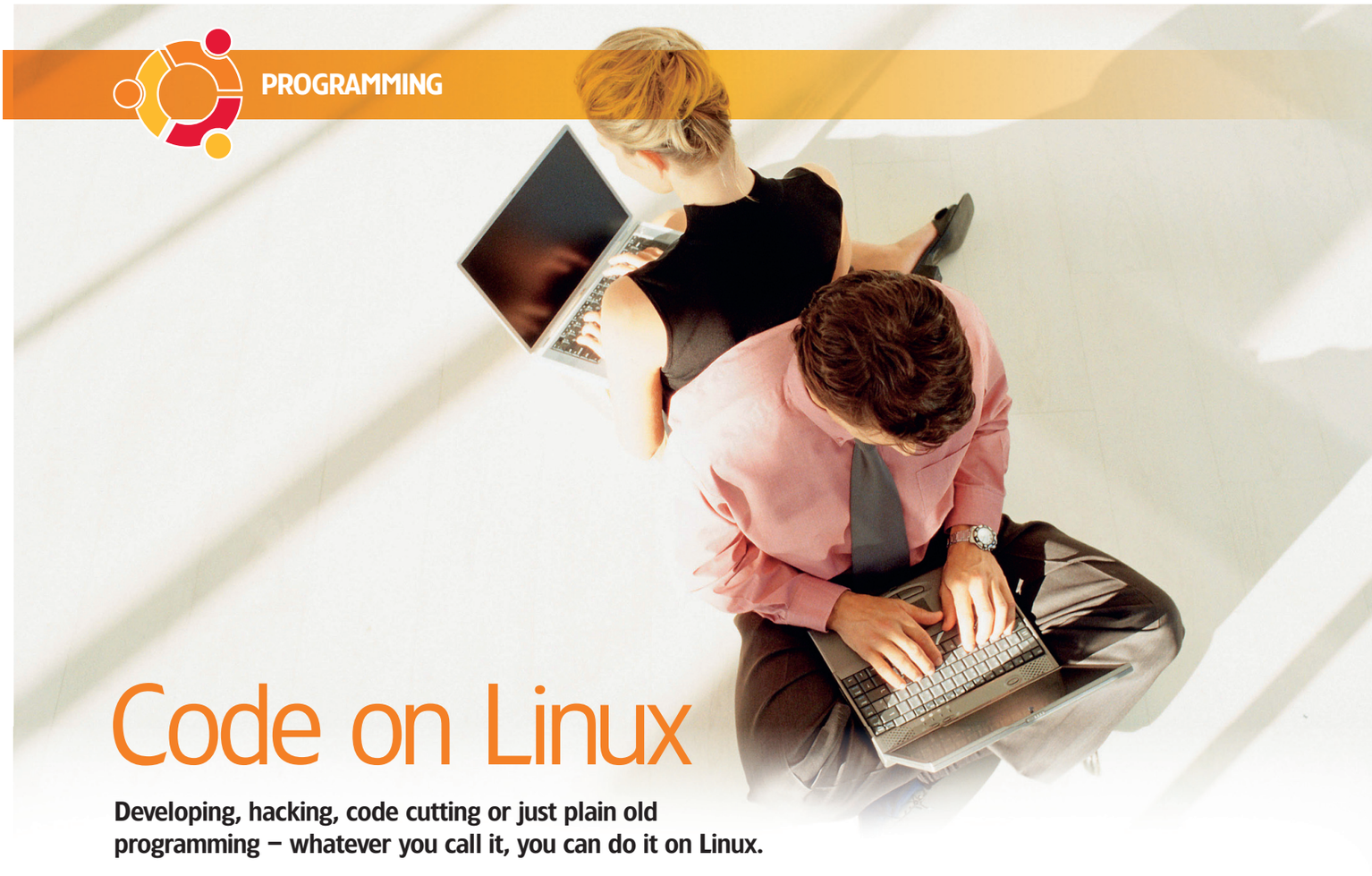


Code on Linux

Developing, hacking, code cutting or just plain old programming – whatever you call it, you can do it on Linux.

MANY PEOPLE consider programming a rather nerdy, obscure art that requires you to tape your glasses together and wear sandals. This couldn't be further from the truth – in fact it's widely practised, and needs to be for open source to flourish. People need access to the code behind programs so they can learn and modify freely.

Although only a few thousand people know enough about system internals to program the Linux kernel, you still get professional quality tools to work on your own projects. Whether you want to program for fun, learn something new or work on a business project, Linux can help you get to grips with a wide range of languages and technologies at no cost.

A BEGINNER'S GUIDE TO CODING TERMS

Array Many variables grouped together into one container variable.
Command interpreter Another name for the command line, or the shell.
Concatenate To join two or more strings together to make one string.
Exponentiation Raising one number to the power of another, eg 10 to the power 3 is 1,000.
Function A discrete and re-usable block of code that can be passed varying output parameters.
Loop Block of code that will run multiple times until a condition is satisfied.
Parameter Variable passed into a function to customise its execution.
Print To output a value – such as a static string or a variable – to the screen.
String A group of characters forming a piece of text (literally: "string of characters").
Subroutine Another word for a function. Also procedure and subprogram.
Syntax Group of rules that define how a programming language needs to be written.
Variable A single data store in a program that is capable of storing custom values during run-time.

WHAT'S ON OFFER

Although some might try to convince you otherwise, there is no one true language for programming. Instead, Linux is a polyglot environment where you can choose the language that fits your needs from its wide selection. Here are the most popular:

- Shell scripting.
- Perl.
- PHP.
- Python.
- C/C++.
- Java.

Most distros support the first five of those out of the box, with Java support varying. Some distros – notably SUSE – install Java as standard, whereas others – notably Fedora – never install Java, leaving you to download it from the website of Sun Microsystems, its creator.

Each language has its own advantages and disadvantages. For example, C++ creates very fast programs, but is very complicated to use, whereas PHP lets you program very quickly, but isn't as flexible as Perl.

Over these four pages we'll be covering shell scripting, Perl and PHP. Don't expect to become a master in any of these technologies – this is really just a sampler to give you a taste of what you can do with Linux. If you find a language you like, the next step is to buy a book about the topic and get into it in more depth.

If you're not familiar with basic programming concepts, such as variables, functions and loops, consult our *Beginner's Guide To Coding Terms*, left.

SHELL SCRIPTING

The most basic form of programming is shell scripting, so named because it uses the *bash* shell as its command interpreter. Shell scripting is quite ugly and limited in what it can do, but it is the simplest form of programming available in Linux and so is the best place to begin. If you have ever written batch files in DOS, shell scripts will be familiar to you.

Shell scripting allows you to run groups of commands in the shell as if you were typing them by hand. The upside of this is that you get to use all the Linux command-line tools as if they were functions, but the downside is that shell scripting has very, very precise syntax that you must adhere to if your scripts are to work. For example, unlike many other languages, *bash* shell scripting does not allow you to put whitespace (ie spaces, new lines and tabs) freely in your code.

Let's look at a very basic script. Open up a text editor such as *GEdit* or *Kate* and type this script, naming it **first.sh**:

```
#!/bin/sh
name="A. Linux User"
echo $name;
```

The first line is known as the shebang, which is an

abbreviation of the 'sharp' and 'bang' names given to the # and ! characters respectively. This specifies which interpreter should be used for this shell script – we have used **/bin/sh**. Nearly every Linux system maps this to *bash*, so this sets up *bash* to interpret our commands.

The next line sets the variable **name** to the string value **A. Linux User**. Note that there is no space either side of the assignment symbol – if you add spacing here it simply won't work. The final line uses the **echo** command to print the value of the **$name** variable, which as you can see has now acquired a dollar sign prefix to signal that it's a variable.

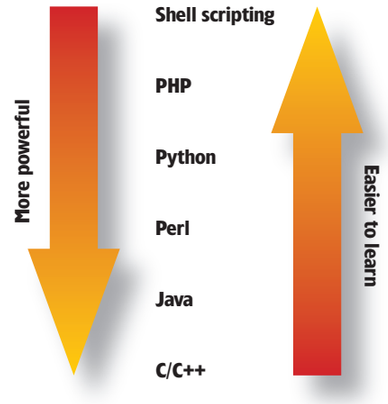
Once you've saved the script and gone back to the shell, run these commands:

```
chmod u+x first.sh
./first.sh
```

This makes the script executable, then runs it. If everything has worked, you should see our variable name being printed out – simple enough.

ACCEPTING PARAMETERS

Any parameters you pass to your script are available in numbered variables, starting from $1, with $0 set to the name of the script that was run. You might think there is little point having $0 because a program will always know its name, but actually this is very useful to have when a program is given multiple names and you need to decide which one the user ran before settings are applied. Along with the numbered dollar variables you also get $#, which contains the number of parameters passed in (excluding $0).



Looking for a language to learn? Linux supports all these and more.

> "Mastery of Perl gives you the coveted title of Perlmonger and firmly places you in the ranks of the Linux elite."

We can use this along with a value comparison to write a simple script that prints a name and age if they are supplied.

Save this next script as **second.sh**:

```
#!/bin/sh
if [ $# -ne 2 ]; then
  echo "You must supply two parameters."
else
  echo "$1 is $2 years old."
fi
```

Line two is now a conditional statement that only evaluates to true if precisely two parameters have been passed in to our script. The formatting is confusing at first: you need **if** followed by a square bracket, then **$#** (the number of parameters passed in), **-ne** for 'not equal', then the value to check against, followed by a closing square bracket and a semi-colon. It's ugly, but once you get the hang of it you'll be fine.

As with other languages, you first specify the action to take if the condition evaluates to true, then use an **else** statement to specify the action to take if the condition evaluates to false. In our script, if two parameters are not provided we print an error message, otherwise we use **$1** and **$2** to print out a short message about someone. To finish the conditional statement, we use **fi** – the reverse of 'if.'

Along with **-ne** there are various other numerical comparisons, such as **-eq** for equals, **-gt** for greater than, **-ge** for greater than or equal to, and **-lt** and **-le** for less than and less than or equal to.

To test whether a file exists you can use **-f**, and to test whether two strings are equal you can use **=**. However, the majority of your *bash* scripting will involve running external programs, and to do that you need to find the backtick key on your keyboard. This is usually to the left of the number 1 on the top of your keyboard (above Tab) and looks like: `.

To execute an external program, just put the command you want inside backticks and place its return value into a variable. For example, if you want to run the *wc* command to count the number of words in a file, you would use **name=`wc somefile.txt`;**. We can use this, along with the new **-f** file check, to write a script that accepts a parameter, checks whether the file exists, and if it does, counts the number of words that it contains. Save this next script as **third.sh**:

```
#!/bin/sh
if [ $# -ne 1 ]; then
  echo "Please provide a parameter.";
else
  if [ -e $1 ]; then
    words=`wc -w $1 | cut -f 1 -d " "`;
    echo "$1 has $words words.";
  else
    echo "$1 does not exist!";
  fi
fi
```

The magic here all happens in the **words=** line, which executes quite a complex command. First it runs **wc -w $1**, which runs the word counting program *wc* and asks it to report only the number of words (**-w**) rather than also counting the number of letters and lines. However, some versions of *wc* will also print out the name of the file it was counting, and we don't want that in our variable. So, the output from **wc** is piped into the *cut* command, which enables you to break text into fields.

In our code, **cut** is given the **-d** parameter along with a pair of quotes with a space in between. This defines the delimiter – what *cut* is using to define individual feeds – as being a space. The **-f 1** parameter means 'we want the first field only.' So, the complete *cut* command breaks our text string up into individual words, then returns the first one – exactly what we need.

PERL

The real power of shell scripting comes about through some of the command-line Linux tools such as *sed* and *awk*, which enable shell scripts to do things that would otherwise be impossible. The problem with this approach is that writing a complex shell script requires mastery of a great number of such tools, and its execution requires Linux to run many different programs.

The solution was to take the functionality of all these basic tools, extend them and roll them up in to a new programming language called Perl, the Practical Extraction and Reporting Language. Perl is a bit of a mongrel programming language in that its design was drawn from so many places that it looks quite confused in places. That said, mastery of Perl gains you the coveted title of Perlmonger, and firmly places you in the ranks of the Linux elite.

Before you start, make sure you have Perl installed. Most distros install it by default, but if you type **perl** at the command line and get 'command not found' you need to use your package manager to install it.

A basic Perl script looks very much like a *bash* shell script, but there are subtle differences. Save this as **first.pl**:

```
#!/usr/bin/perl
$perl = "Great\n";
print $perl;
```

Line one uses the same shebang as with a shell script, but note that we're now using **/usr/bin/perl** as our interpreter rather than **/bin/sh**. Line two sets the variable **$perl**, which has a dollar sign when being set, plus spacing on either side of the equals sign for readability – Perl has no problems handling this. Also note that after the **Great** string comes **\n**, which is known as an escape sequence. This particular escape sequence represents a line break, and without it Perl would not print a new line after **Great** and the user's command prompt would appear on the same line.

ARRAY LOOPS

As you can see, Perl is a logical step up from shell scripting. It's harder but more powerful, and has some truly mind-bending syntax that makes for quick and easy programming. For example, parameters being passed in to the script are stored

in the **ARGV** array (argument values), and because that variable is an array rather than a simple value you need to use an @ symbol rather than a dollar sign for it. So, to print out the first parameter passed to your script, you would use this:

```
print @ARGV[0]
```

The **0** is important because Perl arrays are zero-based, which means the first element is at position 0, the second is at position 1, and so on.

If you want to print out all the parameters passed in to your script, you can use the **foreach** loop and the magic **$_** variable. The combination of these two looks like black magic to outsiders, but we think you will admit that it makes for easy programming – try this out as **second.pl**:

```
#!/usr/bin/perl
foreach(@ARGV) {
  print "$_\n";
}
```

The **foreach** loop goes through each item in an array and, unless told otherwise, places its value into the **$_** variable. We can then print this out inside the loop, along with a new line for easier reading thanks to **\n**. If you want to place the array element into a specific variable, you can use this:

```
#!/usr/bin/perl
foreach $val (@ARGV) {
  print "$val\n";
}
```

To run this script, use **./second.pl First Second Third Fourth**.

SUPER SUBROUTINES

If you want to reuse code in Perl, your best bet is subroutines. These are very easy to use, and by default share variables with the rest of the script. For example, we could modify our script so that it accepted numbers as input and returned those values squared.

Save this script as **third.pl**:

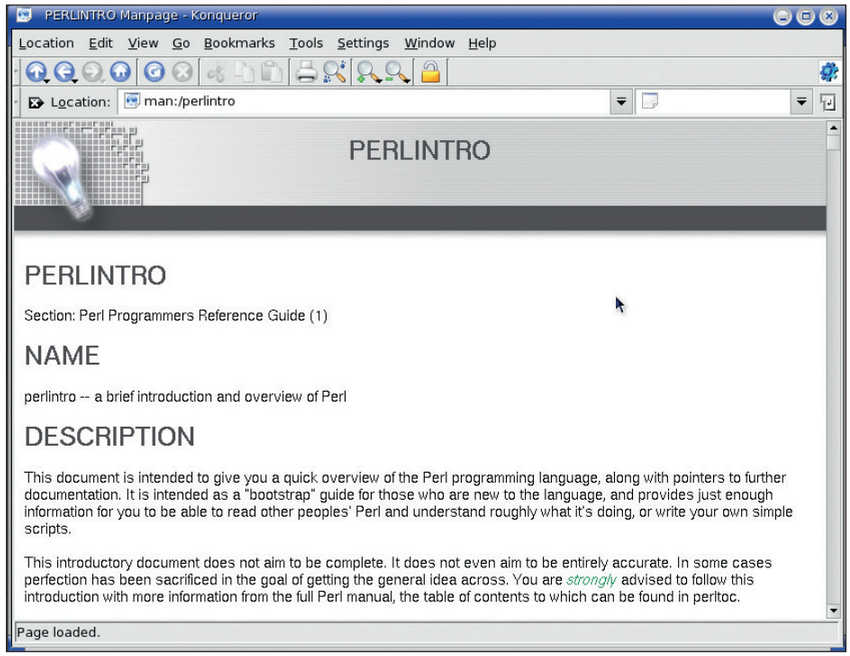
```
#!/usr/bin/perl
sub myfunc {
  return $_ ** 2;
}
foreach (@ARGV) {
  print myfunc . "\n";
}
```

The loop now calls the **myfunc** subroutine, which grabs the **$_** and squares it using the exponentiation operator, **. The result is returned from **myfunc** using the imaginatively titled **return** keyword, then the concatenation operator (a full stop) is used to join the squared number with **\n** so that each number gets a new line.

To run that script, use **./third.pl 1 2 3 4 5**. If you try running it with words rather than numbers you will just get 0, because Perl cannot multiply a string by a string.

In the example above you can see how the **myfunc** subroutine has access to the variables in the rest of the script, and if you play around with it you'll find that any variables set inside **myfunc** are also available in the rest of the script. If you would rather that **myfunc** had its own variables, you should just use the **my** keyword. For instance, **my $number = 6** declared inside **myfunc** would only be available inside the subroutine.

If you are feeling adventurous, you can also use Perl with *Apache* through **mod_perl** (see **http://perl.apache.org**), but most people just turn to PHP.



If you have *Konqueror* installed, loading man:/perlintro will bring up a concise Perl tutorial to help get you started. See our *Where To Go From Here* box on the next page for more assistance.

PHP

Perl has long been the mainstay of Linux script programmers, although over the last five or so years an increasing number of people have been moving to Python for shell scripting. A similar movement has been taking place on the web-scripting front, although this time to PHP, a newer Java-like language that is much easier to learn than Perl and can be integrated very well with both *Apache* and various database systems.

If you followed our *Apache* and PHP installation instructions on page 74 you will already have these two installed and ready to use. If not, go there now and get them installed before you continue. Once you have the **phpinfo()** script working you're ready to get started with some real programming.

Now that you've got everything, let's get started. Save the following code as **first.php** in your **/var/www/html** directory (or in the **public_html** directory for your user):

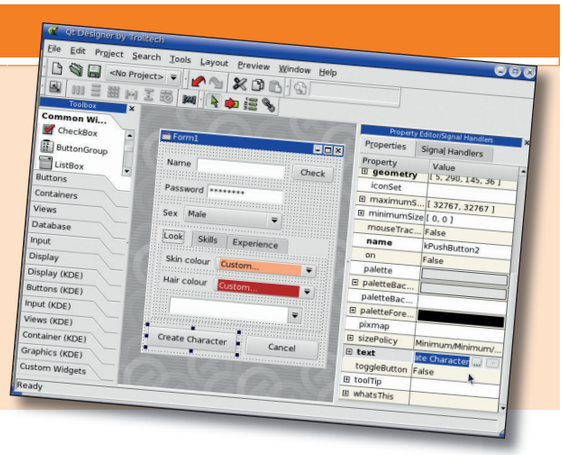
```
This is plain text.<br />
So is this.<br />
<?php
  $name = "J. Random Hacker";
  echo "$name<br />";
?>
Back in plain text again.
```

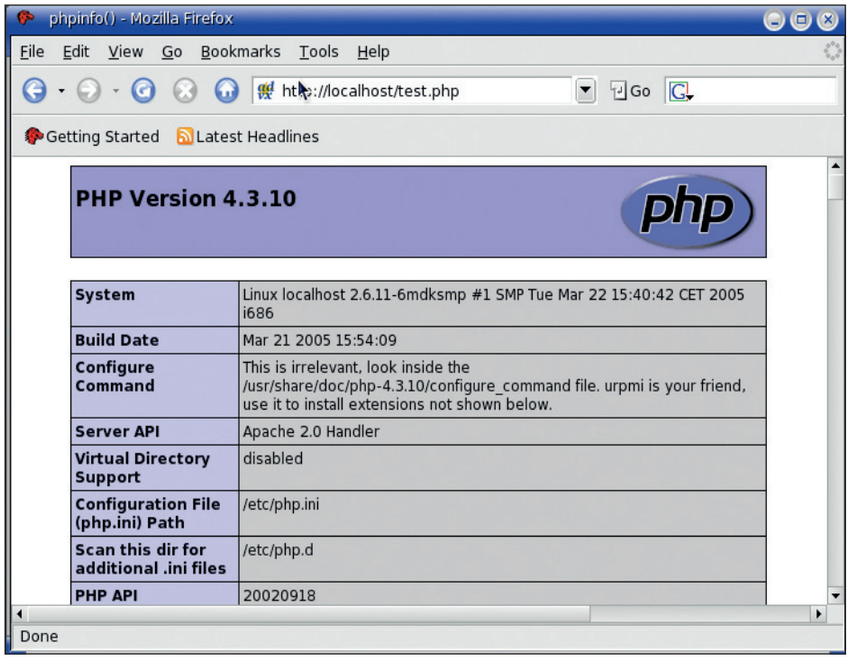
Now open up *Firefox* and load the location **http://localhost/first.php** (or **http://localhost/~yourusername/first.php** if you're doing it the alternative way) – you should see the page printed out. The ability to embed PHP code freely inside HTML is one of PHP's primary benefits. The PHP interpreter operates in HTML mode by default, and just prints everything out, but as soon as you use **<?php** it switches to PHP mode and executes

OR TRY: C

If you're looking for a real challenge, Linux comes complete with C and C++ programming tools that will enable you to create everything from a basic command-line temperature converter to a database-enabled graphical user interface.

Design graphical user interfaces by dragging and dropping what you want using *Qt Designer*.





If you see this screen for your phpinfo() script, you're ready to rock and roll.

"Though PHP is primarily used on the web, extra packages allow it to be used as a command-line interpreter."

everything up until the **?>**, at which point it goes back to HTML mode.

BADGER POPPING

PHP has arrays and loops in much the same way as Perl, although there are noticeably fewer of those 'magic' short scripts involved. This next script creates an array, uses a special function that takes off the last item, then prints it all out – save it as **second.php** in your HTML directory of choice.

```
<?php
  $adverbs = array("Quickly", "Merrily", "Boldly", "Badger");
  $not_adverb = array_pop($adverbs);
  echo "<p><strong>$not_adverb</strong> isn't an adverb!</p>";
  echo "<p><em>These</em> are adverbs...<ul>";
  foreach($adverbs as $adverb) {
    print "<li>$adverb</li>";
  }
  echo "</ul></p>";
?>
```

The **array()** line lets you specify elements to place inside an array by separating each item with a comma. This array, **$adverbs**, then gets passed into the **array_pop()** function, which takes the last element (**"Badger"**) off it and stores it in the **$not_adverb** variable. That variable is then printed out, and we go in to a **foreach** loop to print out the remaining adverbs.

"Badger is not an adverb," he said badgeringly.

Unlike Perl, you must always specify where you want elements to be stored inside a **foreach** variable if you want to use them. In our example we say **$adverbs as $adverb**, which means, 'Loop through the **$adverbs** array and place each element into the **$adverb** element.' The HTML inside and surrounding the loop handles bold, italics, paragraphing and list formatting for us.

We've got just enough room here for one last PHP example, so this time we're going to write a script that accepts a variable from the outside world, checks it, and if successful redirects the user to another page.

Save this as **third.php**:

```
<?php
  if ( isset($_GET["Password"]) ) {
    if ($_GET["Password"] == "nsiucxk") {
      header("Location: secret.php");
      exit;
    } else {
      echo "Oops - wrong password!";
    }
  } else {
    echo "You need to specify a password!";
  }
?>
```

This introduces two new functions, **isset()** and **header()**, and the **$_GET superglobal** array. **$_GET** is a variable set by PHP to contain an array of all the variables that have been sent to the script through the URL. The **isset()** function literally means 'is a variable set', and returns true if the variable exists. So, line two of our code is a conditional statement that checks whether the URL contained the variable **Password**. If it did, we go on to check whether that password was set to **nsiucxk**, and if that condition is also true we use **header()** to tell the web browser to load a different URL.

In this example, **header()** is used to tell the browser to load **secret.php**, and it will do that. However, despite being told to load another URL, the browser will continue loading the current page in the meantime, so to ensure nothing else gets printed out once we've received the correct password we use **exit** to terminate the script.

Although PHP is primarily used on the web, most distros come with extra packages that allow PHP to be used as a command-line interpreter for shell scripting. Look for *php-cli* in your package manager – once that is installed, you should be able to use **#!/usr/bin/php** as your shebang line, or just run scripts through the *php* command, such as **php myscript.php**. ●

WHERE TO GO FROM HERE

Linux Format **frequently runs tutorials on each of these languages, highlighting new and interesting techniques to take your learning to the next level. If you prefer something weightier, there are a number of good books out there:**

- **SHELL PROGRAMMING:**

Unix Shells By Example **by Ellie Quigley (Prentice Hall, ISBN: 0-13-147572-X).**

Unix Shell Programming **by Kochan and Wood (Sams, ISBN: 0-672-32490-3).**

- **PERL PROGRAMMING:**

Anything by O'Reilly.

- **PHP PROGRAMMING:**

PHP 5 Power Programming **by Gutmans *et al* (Prentice Hall, ISBN: 0-13-147149-X).**

PHP and MySQL Web Development **by Welling and Thomson (Sams, ISBN: 0-672-32672-9).**

The Web Programming CD Bookshelf **(O'Reilly, ISBN: 0-596-00510-5).**

- **C AND C++ PROGRAMMING:**

The C++ Programming Language **by Bjarne Stroustrup (Addison-Wesley, ISBN: 0-201-70073-5).**

C++ GUI Programming With Qt 3 **by Blanchette and Summerfield (Prentice Hall, ISBN: 0-13-124072-2).**