Perl 101 (part 7) − CGI Basics

By Vikram Vaswani and Harish Kamath

Table of Contents

Moving On

If you've been paying attention these last few weeks, you should now know enough to write basic Perl programs. And with that task accomplished, Perl 101 now turns its attention to teaching you how to use Perl to generate HTML pages dynamically through server−side CGI scripts.

Since CGI scripts typically involve passing a number of name−value pairs of variables from one page to another, we're first going to introduce you to a new type of variable, and then use that knowledge to build some simple CGI scripts. Keep reading!

Meet Donald Duck

So far, you've used two different types of Perl variables − the scalar and the array. However, unbeknownst to you, Perl also comes with a third type of variable − the hash.

One of the significant features of an array is that array values can only be accessed via a numerical index. This implies that if you need to access an element of the array, you need to first know its exact location in the array. Since this can get complicated with large arrays, Perl offers you a simpler way to access array values, using easy−to−remember "keywords" or "keys".

Thus, a hashes is a species of Perl variable which allows you to define an array of key−value pairs. Take a look:

```
%myhero = ("fname" => "Donald", "lname" => "Duck");
```

The Perl statement above will create a hash named "myhero", which consists of two name−value pairs. The first key is "fname", and it points to the value "Donald", while the second is "lname" and it points to "Duck".

You can also write the statement above like this:

```
%myhero = ("fname", "Donald", "lname", "Duck");
```

Accessing the elements of a hash is equally simple − in the example above, the notation

```
$myhero{"fname"}
```

will return the first value of the hash ("Donald"), while

```
$myhero{"lname"}
```

will return the second value ("Duck").

The rules following hash names are the same as those for scalar and array variables − a hash name begins with a % symbol, followed by an alphabetic character, which may be followed by one or more numbers or letters. Hashes also have their own "space" in Perl.so the variables $duck, @duck and %duck are not treated as one and the same.

Heroes Of The Silver Screen

Here's a simple program that demonstrates how hashes can be used:

```
#!/usr/bin/perl # define a hash %director = ("1995" => "Mel
Gibson", "1996" => "Anthony Minghella", "1997" => "James
Cameron", "1998" => "Steven Spielberg", "1999" => "Sam
Mendes"); # print print "The Best Director Oscar in 1995 went
to $director{1995}\n"; print "The Best Director Oscar in 1996
went to $director{1996}\n"; print "The Best Director Oscar in
1997 went to $director{1997}\n"; print "The Best Director
Oscar in 1998 went to $director{1998}\n"; print "The Best
Director Oscar in 1999 went to $director{1999}\n";
```

And here's the output:

```
The Best Director Oscar in 1995 went to Mel Gibson The Best
Director Oscar in 1996 went to Anthony Minghella The Best
Director Oscar in 1997 went to James Cameron The Best Director
Oscar in 1998 went to Steven Spielberg The Best Director Oscar
in 1999 went to Sam Mendes
```

In the example above, a hash has been used to store a bunch of name−value pairs, and a print() function has been used to display them.

You can also use the alternate hash notation if you prefer.

```
# define a hash %director = ("1995", "Mel Gibson", "1996",
"Anthony Minghella", "1997", "James Cameron", "1998", "Steven
Spielberg", "1999", "Sam Mendes");
```

Open Sesame

The example above demonstrates how hashes can speed up access to specific values in the array. However, if you'd like to access each and every element of the hash in a sequential fashion, things can get hairy.which is why Perl has a few functions designed to simplify that task.

The first of these is the keys() function, which returns a list of all the keys in the specified hash as an array. Here's an example:

```
#!/usr/bin/perl # define a hash %director = ("1995" => "Mel
Gibson", "1996" => "Anthony Minghella", "1997" => "James
Cameron", "1998" => "Steven Spielberg", "1999" => "Sam
Mendes"); # get the names @year = keys(%director); # and use
them in a loop foreach $year(@year) { print "And the Oscar for
Best Director($year) goes to $director{$year}\n"; }
```

And the output is:

```
And the Oscar for Best Director (1995) goes to Mel Gibson And
the Oscar for Best Director (1996) goes to Anthony Minghella
And the Oscar for Best Director (1997) goes to James Cameron
And the Oscar for Best Director (1998) goes to Steven
Spielberg And the Oscar for Best Director (1999) goes to Sam
Mendes
```

In this case, the keys() function returns an array named @year, which looks like this:

```
@year = ("1995", "1996", "1997", "1998", 1999");
```

Once this array has been generated, the "foreach" loop is used to iterate through it and print each name−value pair. And there's a corresponding values() function, which returns.yup, you guessed it, the values from a hash.

```
#!/usr/bin/perl # define a hash %director = ("1995" => "Mel
Gibson", "1996" => "Anthony Minghella", "1997" => "James
Cameron", "1998" => "Steven Spielberg", "1999" => "Sam
Mendes"); # get the names @names = values(%director); print
"The Best Directors on the planet are: \n"; # and use them in
a loop foreach $name (@names) { print "$name \n"; }
```

And the output is:

```
The Best Directors on the planet are: Mel Gibson Anthony
Minghella James Cameron Steven Spielberg Sam Mendes
```

Each() Time The Lights Go Out.

Perl also has the each() function, designed to return a two−element array for each key−value pair in the hash. This comes in particularly useful when iterating through the hash, and is conceptually similar to the foreach() loop. Take a look:

```
#!/usr/bin/perl # define movie hash %film = (1995 =>
'Braveheart', 1996 => 'The English Patient', 1997 =>
'Titanic', 1998 => 'Saving Private Ryan', 1999 => 'American
Beauty'); # define director hash %director = ('Braveheart' =>
'Mel Gibson', 'The English Patient' => 'Anthony Minghella',
'Titanic' => 'James Cameron', 'Saving Private Ryan' => 'Steven
Spielberg', 'American Beauty' => 'Sam Mendes'); # use loop to
iterate through hash while (($year, $filmname) = each %film) {
print "The Oscar for Best Director($year) goes to
$director{$filmname} for $film{$year}\n"; }
```

In the example above, we've created two hashes, one for the movies and years, and the other linking the movies with their directors. Next, we've used the each() function to assign the name−value pairs from the first hash to two variables, $year and $filmname, which are then used to obtain the corresponding director names from the second hash.

Here's the output:

```
The Oscar for Best Director(1995) goes to Mel Gibson for
Braveheart The Oscar for Best Director(1996) goes to Anthony
Minghella for The English Patient The Oscar for Best
Director(1997) goes to James Cameron for Titanic The Oscar for
Best Director(1998) goes to Steven Spielberg for Saving
Private Ryan The Oscar for Best Director(1999) goes to Sam
Mendes for American Beauty
```

Note that Perl allows you to use scalars within the hash notation as well − $director{$filmname} above is an example of this.

And finally, the delete() function allows you to delete a pair of elements from the hash. For example, if you have the hash

```
%film = (1995 => 'Braveheart', 1996 => 'The English Patient',
1997 => 'Titanic', 1998 => 'Saving Private Ryan', 1999 =>
'American Beauty');
```

you can delete the second entry(1996) like this:

```
delete $film{1996};
```

And your hash will then look like this:

```
%film = (1995 => 'Braveheart', 1997 => 'Titanic', 1998 =>
'Saving Private Ryan', 1999 => 'American Beauty');
```

Perl And CGI

You're probably wondering what all this has to do with anything, and why exactly we're torturing you with it at this stage in this series. Well, hashes come in very useful when writing CGI scripts that run off your Web server.and coincidentally, that's just what we're going to be talking about next.

Over the last couple of months, you've spent a great deal of effort understanding variables, loops, file input and output, and string manipulation. The goal of all this: to give you the confidence to use Perl to develop dynamic Web pages. And we're now at the point where you begin applying your hard−won knowledge to some real−life applications.

In non−geek terms, CGI, or the Common Gateway Interface, is a programming environment which allows you to communicate between Web pages and a Web server. So, when you submit a form on a Web site (for example), a CGI program on the server receives your information, saves it to a database or flat file, and dynamically generates an acknowledgment page. The form data is usually submitted is a series of name−value pairs (think hashes!) which is then interpreted and used by the CGI program.

Of course, CGI programs can do much more than this − you can run CGI programs to communicate with a database, read and write flat files, or run processes on the server.

Perl is by far the most common language used for CGI programs. If you're using Apache, you should have the ability to execute CGI scripts out of the box − although you need to keep the following points in mind:

* Your CGI script usually needs to be stored in a particular directory for it to work correctly. This directory is usually the /cgi−bin/ directory off your Web server, although you should check with your Webmaster or system administrator to make sure.

* CGI scripts need to be "mode−executable" under *NIX systems. Simply use the "chmod" command to give your scripts, and the directory they reside in, 0755 permission.

* Many Web servers require that your CGI script end in the file extension .cgi. Again, you'll need to check this with your system administrator.

* Since CGI scripts can run commands on the server, they pose an inherent security risk. You'd be well advised to check that your scripts do not open security holes on your system by running them past an experienced Webmaster before using them in a production environment. The examples we're going to be using are meant for demonstration purposes *only*.

* If you're going to process form data via a CGI script, make sure that your server supports both GET and POST methods of transferring data.

'nuff said. Let's actually write one of these babies.

A Cure For Low Self−Esteem

```
#!/usr/bin/perl print "Content−Type: text/html\n\n"; print
"<html><body><h1>God, I'm good!</h1></body></html>";
```

And when you view this page in your browser (assuming you've placed it in the appropriate place with the appropriate permissions, and that the file is called "good.cgi") by surfing to http://localhost/cgi−bin/good.cgi, you'll see this:

```
God, I'm good!
```

Let's dissect this a bit. The first line of the script is one you'll have to get used to seeing in all your CGI scripts, as it tells the browser how to render the data that follows. In this case, we're telling the browser to render it as HTML text.

Note the two line breaks following the Content−Type statement − forgetting these is one of the most common mistakes Perl newbies make, and it's the cause of a whole slew of error messages.

Our next line is a print() statement which simply outputs some HTML−formatted text − this is what the browser will see.

You can also use variables when generating your page − as the next example demonstrates:

```
#!/usr/bin/perl print "Content−Type: text/html\n\n"; print
"<html><body><h1>Tables</h1>"; print "<table border=2>"; for
($x=1; $x<=5; $x++) { print "<tr><td>Row $x</td></tr>"; }
print "</table></body></html>";
```

GETting Your Form To Work

The CGI environment comes with a set of pre−defined variables that can assist in the task of developing server−side scripts. Here's a list of the important ones:

$ENV{REMOTE_ADDR} The address of the client machine
$ENV{REMOTE_HOST} The host name of the client machine
$ENV{REQUEST_METHOD} The method used by a form to request data
$ENV{HTTP_USER_AGENT} The client browser
$ENV{QUERY_STRING} The query string passed if the GET method is used

Our next few examples will give you some idea of how these, and similar variables, can be used within your scripts.

```
#! /usr/bin/perl $ip = $ENV{REMOTE_ADDR}; $browser =
$ENV{HTTP_USER_AGENT}; print "Content−Type: text/html\n\n";
print "<html><body><font face=Arial>Your IP address is $ip and
your browser is $browser</font></body></html>";
```

And now, when you browse to this page, you'll see some information on your IP address and browser version.

Our next example demonstrates how a Perl/CGI script can be used to process data submitted via a form. Here's the form...

```
<html> <head> <basefont face=Arial> </head> <body> <form
action=http://localhost/readform.cgi method=GET> Enter your
first name: <input type=text length=20 name=name><br> <input
type=submit value=Submit> </form> </body> </html>
```

...and here's the CGI script that receives and processes the data.

```
#!/usr/bin/perl # readform.cgi # reads form data and generates
a page # for GET data if ($ENV{'REQUEST_METHOD'} eq "GET") {
$yourname=$ENV{'QUERY_STRING'}; } # for POST data else {
$yourname = <STDIN>; } # Remove spaces if any $yourname =~
s/\+/ /g; # split form data and store in hash %details = split
(/=/, $yourname); # generate page print "Content−Type:
text/html\n\n"; print "<html><body>"; while (($name, $value) =
each %details) { print "Thank you for your submission,
$value!\n"; } print "</body></html>";
```

The script above will accept data from the HTML form and store it in a variable called $yourname. The manner in which data is submitted by the form(GET or POST) decides the manner in which it is assigned to the variable; the QUERY_STRING environment variable is used to make this decision.

If the text entered into the form contains spaces, the spaces are replaced with + characters when the form is submitted − this needs to be reversed via a regex. For example, if you enter the name "Luke Skywalker" into the form, the URL string will look like this:

http://localhost/cgi−bin/readform.cgi?name=Luke+Skywalker

Next, the name−value pairs are split apart on the basis of the = sign, and are assigned to their respective places in the hash %details. This hash is then used to print the name in the result page, which is also generated by the same script.

Thus, the CGI environment can be used to accept data from one Web page, process it or transfer it to another, and generate a new page containing dynamically−generated output. This is the basis of using CGI to create dynamic Web sites.

In the next issue of Perl 101, we'll delve deeper into CGI, with a look at some simple CGI programs that can be used to track hits on your Web site, or store visitor comments. Don't miss it!