

Chapter 5: Generating HTML Dynamically

There are different operators for string comparison in Perl.

For numeric comparison

```
== (equals)
!= (not equals)
> (greater than)
>= (greater than or equal to)
< (less than or equal to)
<= (less than or equal to)
```

For string comparison

```
eq (equals)
ne (not equals)
gt (greater than)
ge (greater than or equal to)
lt (less than or equal to)
le (less than or equal to)
```

Example:

```
("Hello" eq "hello") false expression
```

Be careful!

```
("Hello" == "hello") true expression -- the operator causes the strings to be placed into numeric context (0 == 0)
```

The Logical Operators are Standard:

```
&& (and - true only if both operands are true)
|| (or - false only if both operands are false)
! (not - negates its operand)
```

The conditionals are standard:

```
if (Boolean expression) {
    block of statements;
}
```

```
if (Boolean expression) {
    block of statements;
}
else {
    block of statements;
}
```

Caution: The curly braces are required in ALL cases

Nested Conditionals are NOT standard:

```
if (Boolean expression) {
    block of statements;
}
elseif (Boolean expression) {
    block of statements;
}
.
.
elseif (Boolean expression) { {
    block of statements;
}
else {
    block of statements;
}
```

The reason for elseif is that the curly braces are mandatory. Hence, the standard else if notation won't work.

Some notes on Boolean variables in Perl:

- There is no Boolean bareword literal value of true (like Java, JavaScript).

- Boolean expressions evaluate to 0 and 1
`$x = ("hello" ne "Hello"); # $x contains 1`
(for true)

`if($x){ ...}` is then equivalent to `if(1){ ...}`

- Strings in Boolean context work as follows:
 - The empty string evaluates to false
 - Any non-empty string evaluates to true

`if ($str) { executes unless $str is empty }`

The **while** loop is standard:

```
while ( Boolean expression ) {  
    block of statements  
}
```

The **for** loop is standard (if you like \$ characters):

```
for ($i=1 ; $i<=100 ; $i++) {  
    block of statements  
}
```

See whileloop.cgi

Arrays

List literal:

```
("a", 'b', 3+4, $x, sqrt(2))
```

Nested lists are **flattened**

```
("a", ('b', 3+4), $x, sqrt(2)) # would be same  
                                # as the above  
                                # list
```

Interesting: Can do multiple scalar variable assignments using list literals

```
($x,$y) = ($y,$x);           # equivalent to  
                                # standard swap
```

- List literals can be assigned to array variables:

```
@names=("frodo","samwise","glorfindel");
```

- The array values are scalar variables

```
print "$names[0]"; # prints frodo
```

```
$names[3] = "shelob"; # add another name  
                      # to the array
```

- The special **\$#names** scalar contains the highest array index, **3** in this case

```
for ($i=0 ; $i <= $#names ; $i++) {  
    print "$names[$i]\n";  
}
```

The **foreach** loop:

```
foreach $item ( @somearray ) {  
    print "$item\n";  
}
```

- The **\$item** variable will go through all the array items in order.
- You can use any scalar variable name (**\$item** is not special)
- Much cleaner and easier to use than a for loop

See links.cgi

Some useful array functions

<code>print @list</code>	prints items in list concatenated together
<code>chomp @list</code>	removes all terminating newline characters from each item in the list, returns total number of characters chomped
<code>chop @list</code>	chops off last character of each item in the list, returns last chopped character
<code>push @array, list</code>	pushes the list onto the back of the array, returns new length
<code>unshift @array, list</code>	pushes the list onto the front of the array, returns new length
<code>pop @array</code>	deletes last element of array, returns the element
<code>shift @array</code>	deletes first element of array, returns the element
<code>join expr, list</code>	returns a string that is the concatenation of all the elements in the list with <code>expr</code> in between each pair of array elements
<code>sort list</code>	
<code>sort <=> list</code>	
<code>splice @array, \$index, \$number</code>	deletes <code>\$number</code> of elements of <code>@array</code> starting at index number <code>\$index</code> , returns the list of deleted elements (A list can be supplied as a fourth argument in which case the list is "spliced" into the array replacing the deleted elements, hence the name of the function.)

We will use the `split` function a lot

```
$str="colon:delimited:list:of:strings";
```

```
@thelist = split(/:/, $str, 3);
```

causes `@thelist` to contain

```
("colon", "delimited", "list")
```

Can use without the third argument to recover all of the delimited

```
@thelist = split(/:/, $str);
```

returns all 5 of the delimited list of strings into `@thelist`.

Hash -- A data structure consisting of key-value pairs

```
%hexColor = (
    keys      values
    "red" => "#ff0000",
    "green" => "#00ff00",
    "blue" => "#0000ff" );
```

- Kind of like an array, but the values are indexed with strings (the keys) instead of integers like with an array.
- A value is accessed using the corresponding key.

```
print $hexColor{'green'};
```

- Again, the value is a scalar, hence the \$

- A list assigned to a hash variable, is treated as a hash.

```
%hexColor = ("red", "#ff0000", "green", "#00ff00", "blue", "#0000ff");
```

- This defines the same hash as using the special hash notation on the previous slide.

- You can also build a hash by direct assignment of the keys and values.

```
$hexColor{"red"}="#ff0000"; $hexColor{"green"}="#00ff00";
$hexColor{"blue"}="#0000ff";
```

- Again, this defines the same hash.

Two useful hash functions

keys %hash	returns the list of keys from %hash
values %hash	returns the list of values from %hash

- They both extract an array from a hash.

```
@k = keys %hexcolor;  
@v = values %hexcolor;
```

Important note: A hash does not preserve the order in which the key-value pairs are assigned to the hash. Thus, two arrays above might not be parallel in the sense of preserving the key-value relationships.

- The **sort** array function is useful to use to iterate over the keys of a hash in alphabetical (ASCII) order.

```
foreach $key (sort keys %hexcolor) {  
    print "$key : $hexcolor{$key}\n";  
}
```

The printed results

```
blue : #0000ff  
green : #00ff00  
red : #ff0000
```

Example:

```
% websites=( 'perl' => 'www.perl.org',  
             'princeton' => 'www.princeton.edu',  
             'amazon' => 'www.amazon.com',  
             'ebay' => 'www.ebay.com' );
```

```
print "<ul>\n";  
foreach $name (keys %websites) {  
    print "<li><a href=\"http://  
$websites{$name}\">$name</a></li>\n";  
}  
print "</ul>\n";
```

Results:

```
<ul>  
<li><a href="http://www.ebay.com">ebay</a></li>  
...  
<li><a href="http://www.amazon.com">amazon</a></li>  
</ul>
```

Two more useful hash functions:

exists \$somehash{ "somekey" }	returns true if somekey exists in %hash
delete \$somehash{ "somekey" }	deletes the entry for somekey from %hash

See topics.cgi

Remember:

- Array variable

```
@ a = ('value1', ..., 'valueN');
```

- Individual array values are scalar variables indexed with integers 0,1,...

```
$a[0]
```

- Hash variable

```
% h = ('key1' => 'value1',  
      ...,  
      'keyN' => 'valueN');
```

- Individual hash values are scalar variables indexed with strings (the keys). The order in which the hash actually stores the keys is unpredictable.

```
$h{'key1'}
```

- Scalar, array and hash variables have different namespaces. Thus, the following would all be distinct variables in a program:

```
$x
```

```
@ x
```

```
% x
```

- Moreover, the following scalars would also be distinct variables in a program.

```
$x
```

```
$x[i] # array value
```

```
$x{"key"} # hash value
```

- Perl functions are called sub routines.

```
sub functionName {  
    statements executed by the subroutine  
}
```

- We will just call them functions.
- Self defined functions should be called using the syntax

```
&functionName()
```

- The parentheses are optional when there are no parameters.

```
&functionName
```

Example of a void (non-return) function:

```
sub printWebPageFooter {  
    print 'This web page was designed by  
    <a href="mailto:me@myaddress.com">  
    send me mail</a>', "\n";  
}
```

Example call to the function:

```
&printWebPageFooter;
```

- Built-in functions are called without the & character.

```
$x = sqrt 16;
@timeparts = localtime;
```

- The built-in local time function returns an array of 9 different time parts. We could call the function like this:

```
($sec, $min, $hour, $mday, $mon, $year, $yday, $isdst) =
localtime;
```

- So, in the first function call `$timeparts[2]` would contain the `$hour` component, for example.

Two points:

- Perl functions can **return** values as you would expect.
- The **my** key word makes variables local to functions.

Example function:

```
sub currentDate {
    my @timeParts=localtime;
    my ($day, $month, $year) =
($timeParts[3], $timeParts[4], $timeParts[5]);
    return ($month+1)."/".$day."/".($year+1900);
}
```

Example use:

```
print "<title>Stock Quotes for ", &currentDate, "</title>\n";
```

See: `currentDate.pl`

Important: Functions can't be interpolated inside double quotes or (print blocks).

Local variables using `my` work as you would expect:

Example:

```
$x=1;
&noChange;
print "$x";

sub noChange {
my $x=2;
}
```

Printed result:

1

(without the `my` keyword, 2 would have been printed)

Passing arguments to Perl functions:

- The built-in array `@_` (the arguments array) receives all parameters in order

Example function call:

```
$domain = "www.cknuckles.com";
$text = "Web applications site";
&makeLink($domain, $text);
```

The function:

```
sub makeLink{
    print "<a href=\"http://$_[0]\">$_[1]</a>";
}
```

The output:

```
<a href="http://www.cknuckles.com">Web applications site</a>
```

In Perl, ALL parameters are passed by reference.

Example:

```
$x=1;
&change($x);

sub change {
    $_[0] = 2;    # changes the
                  # global variable $x
}
```

Potential Error:

```
&change("hello"); # this would give an error (attempt to
alter a ready only value) since $_[0] is a reference to a string
literal.
```

A good practice in Perl to help avoid that problem (and so you don't have to use all the funky variables like \$_[0]) is to simply transfer the arguments array into local variables.

```
sub makeLink{
    my ($domain, $text) = @_;
    print "<a href=\"http:// $domain\">$text
</a>";
}
```

Because the arguments array (like all Perl arrays) is not of fixed size, a function can be constructed so that it takes an arbitrary number of parameters (without overloading it).

Example:

```
sub makeList {
    print "<ul>\n";
    foreach $item (@_) {
        print "<li>$item</li>\n";
    }
    print "</ul>\n";
}
```

Sample calls:

```
&makeList("apples", "oranges");
&makeList("apples", "oranges", "bananas");
```

See makeList.pl

File Operations in Perl:

Access Mode	Qualifier
read	open (FILEHANDLE, filename)
read	open (FILEHANDLE, "<filename")
write (overwrite)	open (FILEHANDLE, ">filename")
append	open (FILEHANDLE, ">>filename")

Example: Open for appending.

```
open (FILEHANDLE, ">>data.txt");
    print a line to the file
close (FILEHANDLE);
```

Strategies for reading data from a file:

- Grab one line at a time manually as needed:

```
$line1 = <FILEHANDLE>;  
$line2 = <FILEHANDLE>;  
$line3 = <FILEHANDLE>;
```

...

- Process it one line at a time in a loop:

```
while($line = <FILEHANDLE>) {  
    do something with each $line  
}
```

- Read the whole file into an array with one line of code and use as needed later.

```
@all_lines = <FILEHANDLE>;
```



Example: Read this text file into a hash.

```
$dataDir = "directory/path/"; # to the directory  
                                # containing websites.txt  
  
% websites=(); # empty hash  
open(INFILE, "$dataDir"."websites.txt");  
while($line = <INFILE>) {  
    chomp $line;  
    ($name, $address) = split(/=/, $line);  
    $websites{$name} = $address;  
}  
close(INFILE);
```

Error potential when reading files in CGI programs:

- Permissions: need 644 (the default for ftp transfer)
- File does not exist when opening for reading (file names are case sensitive on Unix/Linux)

Note: When opening for write or append, the file will be created if it does not exist. No big deal if you are running a non-CGI Perl program, since you are the user creating the file.

But for CGI programs, the file is created at the request of the Web server software, which runs under owner *nobody*. Thus, the file's owner is nobody. That means you can only read it even if it is in your account!

- In **non-CGI programs**, the standard way to detect failed file operations is

```
open(FILEHANDLE, "somefile.txt") or die "failed file access on somefile.txt";
```

- On failure, the **die** command writes its string argument to the **standard error channel** -- the command line or IDE output window.

- If you set a descriptive **die** string for each file open operation, you can easily debug programs because you know what went wrong.

- **Problem:** In CGI programs, the standard error channel goes into a log file kept by the Web server software. Thus, you won't see the message from the **die** command.

- **The solution:** Send an error Web page back to the Web browser when a file operation fails.

- **Advantages:**

- You can easily debug your programs when a file operation is causing it not to work right.
- Commercial Web sites can send descriptive error pages back to the Web browser (instead of an incomplete page, which is often the result of a failed file operation).

The `&errorPage` function we will use throughout this book.

```
sub errorPage {
    my $message = $_[0];

    print<<ALL;
    <html><head><title>Server Error</title></head>
    <body><h2>Server Error Encountered</h2>
    $message
    If the problem persists, please notify the
    <a href="mailto:x\@y.com">webmaster</a>.
    </body></html>
ALL
    exit; # terminate program
        # CRUCIAL so that the program does
        # not print two complete Web pages
}
```

See `links2.cgi`

- In the previous example, a failed file operation would cause a broken Web page (a list whose list items contain broken links). Thus, sending an error page is certainly appropriate.

- However, in some cases, a failed file operation does not warrant an error page. Rather, a failed file operation can be compensated for in the Web page generated by the program.

- A perfect example is a hit counter, where a text file keeps a record of the number of past hits to the program. Upon a failed file operation, it's better to just not report the current hit total than to abort (`exit`) the program and print an error page.

See `hit.cgi`

External files containing source code.

- The following statement imports Perl code contained in a separate file:

```
require "path/to/somefile.lib";
```

- If the external file contains stand-alone statements (not included in a function), those statements will be executed as part of the program which imports (requires) the external file.

- Usually, such external files are used only to contain useful libraries of functions, hence the `.lib` file extension we used.

- A failed attempt to import (require) an external file is usually fatal to a Perl program. (In contrast to a failed open file operation.)

```
# Auxiliary file containing:
# &errorPage, &someOtherFunction

#####
sub errorPage {
  blagh, blagh, blagh, ...
}
#####

#####
sub someOtherFunction {
  blagh, blagh, blagh, ...
}
#####

1; # CRUCIAL to return true at the BOTTOM
      # of an external source file
```