NAME

HTML::Template – Perl module to use HTML–like templating language

SYNOPSIS

First you make a template – this is just a normal HTML file with a few extra tags, the simplest being `<TMPL_VAR>`

For example, test.tmpl:

```
<html>
<head><title>Test Template</title></head>
<body>
My Home Directory is <TMPL_VAR NAME=HOME>
<p>
My Path is set to <TMPL_VAR NAME=PATH>
</body>
</html>
```

Now you can use it in a small CGI program:

```
#!/usr/bin/perl -w
use HTML::Template;

# open the html template
my $template = HTML::Template->new(filename => 'test.tmpl');

# fill in some parameters
$template->param(HOME => $ENV{HOME});
$template->param(PATH => $ENV{PATH});

# send the obligatory Content-Type and print the template output
print "Content-Type: text/html\n\n", $template->output;
```

If all is well in the universe this should show something like this in your browser when visiting the CGI:

```
My Home Directory is /home/some/directory
My Path is set to /bin;/usr/bin
```

DESCRIPTION

This module attempts to make using HTML templates simple and natural. It extends standard HTML with a few new HTML-esque tags – `<TMPL_VAR>` `<TMPL_LOOP>`, `<TMPL_INCLUDE>`, `<TMPL_IF>`, `<TMPL_ELSE>` and `<TMPL_UNLESS>`. The file written with HTML and these new tags is called a template. It is usually saved separate from your script – possibly even created by someone else! Using this module you fill in the values for the variables, loops and branches declared in the template. This allows you to separate design – the HTML – from the data, which you generate in the Perl script.

This module is licensed under the same terms as Perl. See the LICENSE section below for more details.

TUTORIAL

If you're new to HTML::Template, I suggest you start with the introductory article available on Perl Monks:

```
http://www.perlmonks.org/?node_id=65642
```

FAQ

Please see HTML::Template::FAQ

MOTIVATION

It is true that there are a number of packages out there to do HTML templates. On the one hand you have things like HTML::Embperl which allows you freely mix Perl with HTML. On the other hand lie home-grown variable substitution solutions. Hopefully the module can find a place between the two.

One advantage of this module over a full HTML::Embperl–esque solution is that it enforces an important divide – design and programming. By limiting the programmer to just using simple variables and loops in the HTML, the template remains accessible to designers and other non-perl people. The use of HTML-esque syntax goes further to make the format understandable to others. In the future this similarity could be used to extend existing HTML editors/analyzers to support HTML::Template.

An advantage of this module over home-grown tag-replacement schemes is the support for loops. In my work I am often called on to produce tables of data in html. Producing them using simplistic HTML templates results in programs containing lots of HTML since the HTML itself cannot represent loops. The introduction of loop statements in the HTML simplifies this situation considerably. The designer can layout a single row and the programmer can fill it in as many times as necessary – all they must agree on is the parameter names.

For all that, I think the best thing about this module is that it does just one thing and it does it quickly and carefully. It doesn't try to replace Perl and HTML, it just augments them to interact a little better. And it's pretty fast.

THE TAGS

TMPL_VAR

```
    <TMPL_VAR NAME="PARAMETER_NAME">
```

The `<TMPL_VAR>` tag is very simple. For each `<TMPL_VAR>` tag in the template you call:

```
    $template->param(PARAMETER_NAME => "VALUE")
```

When the template is output the `<TMPL_VAR>` is replaced with the VALUE text you specified. If you don't set a parameter it just gets skipped in the output.

You can also specify the value of the parameter as a code reference in order to have "lazy" variables. These sub routines will only be referenced if the variables are used. See "LAZY VALUES" for more information.

Attributes

The following "attributes" can also be specified in template var tags:

- escape

  This allows you to escape the value before it's put into the output.

  This is useful when you want to use a TMPL_VAR in a context where those characters would cause trouble. For example:

  ```
      <input name=param type=text value="<TMPL_VAR PARAM>">
  ```

  If you called `param()` with a value like `sam"my` you'll get in trouble with HTML's idea of a double-quote. On the other hand, if you use `escape=html`, like this:

  ```
      <input name=param type=text value="<TMPL_VAR PARAM ESCAPE=HTML>">
  ```

  You'll get what you wanted no matter what value happens to be passed in for param.

  The following escape values are supported:

  - html

    Replaces the following characters with their HTML entity equivalent: `&`, `"`, `'`, <, >

  - js

    Escapes (with a backslash) the following characters: `\`, `'`, `"`, `\n`, `\r`

  - url

    URL escapes any ASCII characters except for letters, numbers, _, `.` and `-`.

  - none

    Performs no escaping. This is the default, but it's useful to be able to explicitly turn off escaping if you are using the `default_escape` option.

- default

  With this attribute you can assign a default value to a variable. For example, this will output "the devil gave me a taco" if the `who` variable is not set.

  ```
  <TMPL_VAR WHO DEFAULT="the devil"> gave me a taco.
  ```

TMPL_LOOP

```
<TMPL_LOOP NAME="LOOP_NAME"> ... </TMPL_LOOP>
```

The `<TMPL_LOOP>` tag is a bit more complicated than `<TMPL_VAR>`. The `<TMPL_LOOP>` tag allows you to delimit a section of text and give it a name. Inside this named loop you place `<TMPL_VAR>`s. Now you pass to `param()` a list (an array ref) of parameter assignments (hash refs) for this loop. The loop iterates over the list and produces output from the text block for each pass. Unset parameters are skipped. Here's an example:

In the template:

```
<TMPL_LOOP NAME=EMPLOYEE_INFO>
   Name: <TMPL_VAR NAME=NAME> <br>
   Job:  <TMPL_VAR NAME=JOB>  <p>
</TMPL_LOOP>
```

In your Perl code:

```
$template->param(
    EMPLOYEE_INFO => [{name => 'Sam', job => 'programmer'}, {name => 'Steve',
);
print $template->output();
```

The output is:

```
Name: Sam
Job: programmer

Name: Steve
Job: soda jerk
```

As you can see above the `<TMPL_LOOP>` takes a list of variable assignments and then iterates over the loop body producing output.

Often you'll want to generate a `<TMPL_LOOP>`'s contents programmatically. Here's an example of how this can be done (many other ways are possible!):

```
# a couple of arrays of data to put in a loop:
my @words     = qw(I Am Cool);
my @numbers   = qw(1 2 3);
my @loop_data = ();              # initialize an array to hold your loop

while (@words and @numbers) {
    my %row_data;      # get a fresh hash for the row data

    # fill in this row
    $row_data{WORD}   = shift @words;
    $row_data{NUMBER} = shift @numbers;

    # the crucial step - push a reference to this row into the loop!
    push(@loop_data, \%row_data);
}

# finally, assign the loop data to the loop param, again with a reference:
$template->param(THIS_LOOP => \@loop_data);
```

The above example would work with a template like:

```
<TMPL_LOOP NAME="THIS_LOOP">
  Word: <TMPL_VAR NAME="WORD">
  Number: <TMPL_VAR NAME="NUMBER">

</TMPL_LOOP>
```

It would produce output like:

```
Word: I
Number: 1

Word: Am
Number: 2

Word: Cool
Number: 3
```

<TMPL_LOOP>s within <TMPL_LOOP>s are fine and work as you would expect. If the syntax for the `param()` call has you stumped, here's an example of a param call with one nested loop:

```
$template->param(
    LOOP => [
        {
            name      => 'Bobby',
            nicknames => [{name => 'the big bad wolf'}, {name => 'He-Man'}],
        },
    ],
);
```

Basically, each <TMPL_LOOP> gets an array reference. Inside the array are any number of hash references. These hashes contain the name=>value pairs for a single pass over the loop template.

Inside a <TMPL_LOOP>, the only variables that are usable are the ones from the <TMPL_LOOP>. The variables in the outer blocks are not visible within a template loop. For the computer-science geeks among you, a <TMPL_LOOP> introduces a new scope much like a perl subroutine call. If you want your variables to be global you can use `global_vars` option to `new()` described below.

TMPL_INCLUDE

```
<TMPL_INCLUDE NAME="filename.tmpl">
```

This tag includes a template directly into the current template at the point where the tag is found. The included template contents are used exactly as if its contents were physically included in the master template.

The file specified can be an absolute path (beginning with a '/' under Unix, for example). If it isn't absolute, the path to the enclosing file is tried first. After that the path in the environment variable HTML_TEMPLATE_ROOT is tried, if it exists. Next, the "path" option is consulted, first as-is and then with HTML_TEMPLATE_ROOT prepended if available. As a final attempt, the filename is passed to `open()` directly. See below for more information on HTML_TEMPLATE_ROOT and the `path` option to `new()`.

As a protection against infinitely recursive includes, an arbitrary limit of 10 levels deep is imposed. You can alter this limit with the `max_includes` option. See the entry for the `max_includes` option below for more details.

TMPL_IF

```
<TMPL_IF NAME="PARAMETER_NAME"> ... </TMPL_IF>
```

The <TMPL_IF> tag allows you to include or not include a block of the template based on the value of a given parameter name. If the parameter is given a value that is true for Perl – like '1' – then the block is included in the output. If it is not defined, or given a false value − like '0' − then it is skipped. The parameters are specified the same way as with `<TMPL_VAR>`.

Example Template:

```
<TMPL_IF NAME="BOOL">
  Some text that only gets displayed if BOOL is true!
</TMPL_IF>
```

Now if you call `$template->param(BOOL => 1)` then the above block will be included by output.

`<TMPL_IF> </TMPL_IF>` blocks can include any valid HTML::Template construct − `VAR`s and `LOOP`s and other `IF`/`ELSE` blocks. Note, however, that intersecting a `<TMPL_IF>` and a `<TMPL_LOOP>` is invalid.

```
Not going to work:
<TMPL_IF BOOL>
  <TMPL_LOOP SOME_LOOP>
</TMPL_IF>
  </TMPL_LOOP>
```

If the name of a `<TMPL_LOOP>` is used in a `<TMPL_IF>`, the `IF` block will output if the loop has at least one row. Example:

```
<TMPL_IF LOOP_ONE>
  This will output if the loop is not empty.
</TMPL_IF>

<TMPL_LOOP LOOP_ONE>
  ....
</TMPL_LOOP>
```

WARNING: Much of the benefit of HTML::Template is in decoupling your Perl and HTML. If you introduce numerous cases where you have `TMPL_IF`s and matching Perl `if`s, you will create a maintenance problem in keeping the two synchronized. I suggest you adopt the practice of only using `TMPL_IF` if you can do so without requiring a matching `if` in your Perl code.

TMPL_ELSE

```
<TMPL_IF NAME="PARAMETER_NAME"> ... <TMPL_ELSE> ... </TMPL_IF>
```

You can include an alternate block in your `<TMPL_IF>` block by using `<TMPL_ELSE>`. NOTE: You still end the block with `</TMPL_IF>`, not `</TMPL_ELSE>`!

```
 Example:
  <TMPL_IF BOOL>
    Some text that is included only if BOOL is true
  <TMPL_ELSE>
    Some text that is included only if BOOL is false
  </TMPL_IF>
```

TMPL_UNLESS

```
<TMPL_UNLESS NAME="PARAMETER_NAME"> ... </TMPL_UNLESS>
```

This tag is the opposite of `<TMPL_IF>`. The block is output if the `PARAMETER_NAME` is set false or not defined. You can use `<TMPL_ELSE>` with `<TMPL_UNLESS>` just as you can with `<TMPL_IF>`.

```
Example:
<TMPL_UNLESS BOOL>
  Some text that is output only if BOOL is FALSE.
<TMPL_ELSE>
  Some text that is output only if BOOL is TRUE.
</TMPL_UNLESS>
```

If the name of a `<TMPL_LOOP>` is used in a `<TMPL_UNLESS>`, the `<UNLESS>` block output if the loop has zero rows.

```
<TMPL_UNLESS LOOP_ONE>
  This will output if the loop is empty.
</TMPL_UNLESS>

<TMPL_LOOP LOOP_ONE>
  ....
</TMPL_LOOP>
```

NOTES

HTML::Template's tags are meant to mimic normal HTML tags. However, they are allowed to "break the rules". Something like:

```
<img src="<TMPL_VAR IMAGE_SRC>">
```

is not really valid HTML, but it is a perfectly valid use and will work as planned.

The `NAME=` in the tag is optional, although for extensibility's sake I recommend using it. Example − `<TMPL_LOOP LOOP_NAME>` is acceptable.

If you're a fanatic about valid HTML and would like your templates to conform to valid HTML syntax, you may optionally type template tags in the form of HTML comments. This may be of use to HTML authors who would like to validate their templates' HTML syntax prior to HTML::Template processing, or who use DTD-savvy editing tools.

```
<!-- TMPL_VAR NAME=PARAM1 -->
```

In order to realize a dramatic savings in bandwidth, the standard (non-comment) tags will be used throughout this documentation.

METHODS

new

Call `new()` to create a new Template object:

```
my $template = HTML::Template->new(
    filename => 'file.tmpl',
    option   => 'value',
);
```

You must call `new()` with at least one `name` = value> pair specifying how to access the template text. You can use `filename => 'file.tmpl'` to specify a filename to be opened as the template. Alternately you can use:

```
my $t = HTML::Template->new(
    scalarref => $ref_to_template_text,
    option    => 'value',
);
```

and

```
my $t = HTML::Template->new(
    arrayref => $ref_to_array_of_lines,
    option   => 'value',
);
```

These initialize the template from in-memory resources. In almost every case you'll want to use the filename parameter. If you're worried about all the disk access from reading a template file just use mod_perl and the cache option detailed below.

You can also read the template from an already opened filehandle, either traditionally as a glob or as a FileHandle:

```
    my $t = HTML::Template->new(filehandle => *FH, option => 'value');
```

The four `new()` calling methods can also be accessed as below, if you prefer.

```
    my $t = HTML::Template->new_file('file.tmpl', option => 'value');

    my $t = HTML::Template->new_scalar_ref($ref_to_template_text, option => 'valu

    my $t = HTML::Template->new_array_ref($ref_to_array_of_lines, option => 'valu

    my $t = HTML::Template->new_filehandle($fh, option => 'value');
```

And as a final option, for those that might prefer it, you can call new as:

```
    my $t = HTML::Template->new(
        type   => 'filename',
        source => 'file.tmpl',
    );
```

Which works for all three of the source types.

If the environment variable `HTML_TEMPLATE_ROOT` is set and your filename doesn't begin with "/", then the path will be relative to the value of c<HTML_TEMPLATE_ROOT>.

Example – if the environment variable `HTML_TEMPLATE_ROOT` is set to */home/sam* and I call `HTML::Template->new()` with filename set to "sam.tmpl", HTML::Template will try to open */home/sam/sam.tmpl* to access the template file. You can also affect the search path for files with the `path` option to `new()` – see below for more information.

You can modify the Template object's behavior with `new()`. The options are available:

Error Detection Options

- die_on_bad_params

  If set to 0 the module will let you call:

  ```
      $template->param(param_name => 'value')
  ```

  even if 'param_name' doesn't exist in the template body. Defaults to 1.

- force_untaint

  If set to 1 the module will not allow you to set unescaped parameters with tainted values. If set to 2 you will have to untaint all parameters, including ones with the escape attribute. This option makes sure you untaint everything so you don't accidentally introduce e.g. cross-site-scripting (XSS) vulnerabilities. Requires taint mode. Defaults to 0.

- strict – if set to 0 the module will allow things that look like they might be TMPL_* tags to get by without dieing. Example:

  ```
      <TMPL_HUH NAME=ZUH>
  ```

  Would normally cause an error, but if you call new with `strict => 0` HTML::Template will ignore it. Defaults to 1.

- vanguard_compatibility_mode

  If set to 1 the module will expect to see `<TMPL_VAR>`s that look like `%NAME%` in addition to the standard syntax. Also sets `die_on_bad_params = 0>`. If you're not at Vanguard Media trying to use an old format template don't worry about this one. Defaults to 0.

Caching Options

- cache

  If set to 1 the module will cache in memory the parsed templates based on the filename parameter, the modification date of the file and the options passed to `new()`. This only applies to templates opened with the filename parameter specified, not scalarref or arrayref templates. Caching also looks at the modification times of any files included using `<TMPL_INCLUDE>` tags, but again, only if the template is opened with filename parameter.

This is mainly of use in a persistent environment like Apache/mod_perl. It has absolutely no benefit in a normal CGI environment since the script is unloaded from memory after every request. For a cache that does work for a non-persistent environment see the `shared_cache` option below.

My simplistic testing shows that using cache yields a 90% performance increase under mod_perl. Cache defaults to 0.

- shared_cache

  If set to 1 the module will store its cache in shared memory using the IPC::SharedCache module (available from CPAN). The effect of this will be to maintain a single shared copy of each parsed template for all instances of HTML::Template on the same machine to use. This can be a significant reduction in memory usage in an environment with a single machine but multiple servers. As an example, on one of our systems we use 4MB of template cache and maintain 25 httpd processes − shared_cache results in saving almost 100MB! Of course, some reduction in speed versus normal caching is to be expected. Another difference between normal caching and shared_cache is that shared_cache will work in a non-persistent environment (like normal CGI) − normal caching is only useful in a persistent environment like Apache/mod_perl.

  By default HTML::Template uses the IPC key 'TMPL' as a shared root segment (0x4c504d54 in hex), but this can be changed by setting the `ipc_key new()` parameter to another 4−character or integer key. Other options can be used to affect the shared memory cache correspond to IPC::SharedCache options − `ipc_mode`, `ipc_segment_size` and `ipc_max_size`. See IPC::SharedCache for a description of how these work − in most cases you shouldn't need to change them from the defaults.

  For more information about the shared memory cache system used by HTML::Template see IPC::SharedCache.

- double_cache

  If set to 1 the module will use a combination of `shared_cache` and normal cache mode for the best possible caching. Of course, it also uses the most memory of all the cache modes. All the same ipc_* options that work with `shared_cache` apply to `double_cache` as well. Defaults to 0.

- blind_cache

  If set to 1 the module behaves exactly as with normal caching but does not check to see if the file has changed on each request. This option should be used with caution, but could be of use on high-load servers. My tests show `blind_cache` performing only 1 to 2 percent faster than cache under mod_perl.

  NOTE: Combining this option with shared_cache can result in stale templates stuck permanently in shared memory!

- file_cache

  If set to 1 the module will store its cache in a file using the Storable module. It uses no additional memory, and my simplistic testing shows that it yields a 50% performance advantage. Like `shared_cache`, it will work in a non-persistent environments (like CGI). Default is 0.

  If you set this option you must set the `file_cache_dir` option. See below for details.

  NOTE: Storable uses `flock()` to ensure safe access to cache files. Using `file_cache` on a system or filesystem (like NFS) without `flock()` support is dangerous.

- file_cache_dir

  Sets the directory where the module will store the cache files if `file_cache` is enabled. Your script will need write permissions to this directory. You'll also need to make sure the sufficient space is available to store the cache files.

- file_cache_dir_mode

  Sets the file mode for newly created `file_cache` directories and subdirectories. Defaults to "0700" for security but this may be inconvenient if you do not have access to the account running the webserver.

- double_file_cache

  If set to 1 the module will use a combination of `file_cache` and normal `cache` mode for the best possible caching. The file_cache_* options that work with file_cache apply to `double_file_cache` as well. Defaults to 0.

- cache_lazy_vars

  The option tells HTML::Template to cache the values returned from code references used for `TMPL_VAR`s. See "LAZY VALUES" for details.

- cache_lazy_loops

  The option tells HTML::Template to cache the values returned from code references used for `TMPL_LOOP`s. See "LAZY VALUES" for details.

Filesystem Options

- path

  You can set this variable with a list of paths to search for files specified with the `filename` option to `new()` and for files included with the `<TMPL_INCLUDE>` tag. This list is only consulted when the filename is relative. The `HTML_TEMPLATE_ROOT` environment variable is always tried first if it exists. Also, if `HTML_TEMPLATE_ROOT` is set then an attempt will be made to prepend `HTML_TEMPLATE_ROOT` onto paths in the path array. In the case of a `<TMPL_INCLUDE>` file, the path to the including file is also tried before path is consulted.

  Example:

  ```
  my $template = HTML::Template->new(
      filename => 'file.tmpl',
      path     => ['/path/to/templates', '/alternate/path'],
  );
  ```

  NOTE: the paths in the path list must be expressed as UNIX paths, separated by the forward-slash character ('/').

- search_path_on_include

  If set to a true value the module will search from the top of the array of paths specified by the path option on every `<TMPL_INCLUDE>` and use the first matching template found. The normal behavior is to look only in the current directory for a template to include. Defaults to 0.

- utf8

  Setting this to true tells HTML::Template to treat your template files as UTF–8 encoded. This will apply to any file's passed to `new()` or any included files. It won't do anything special to scalars templates passed to `new()` since you should be doing the encoding on those yourself.

  ```
  my $template = HTML::Template->new(
      filename => 'umlauts_are_awesome.tmpl',
      utf8     => 1,
  );
  ```

  Most templates are either ASCII (the default) or UTF–8 encoded Unicode. But if you need some other encoding other than these 2, look at the `open_mode` option.

NOTE: The `utf8` and `open_mode` options cannot be used at the same time.

- open_mode

  You can set this option to an opening mode with which all template files will be opened.

  For example, if you want to use a template that is UTF−16 encoded unicode:

  ```
  my $template = HTML::Template->new(
      filename  => 'file.tmpl',
      open_mode => '<:encoding(UTF-16)',
  );
  ```

  That way you can force a different encoding (than the default ASCII or UTF−8), CR/LF properties etc. on the template files. See PerlIO for details.

  NOTE: this only works in perl 5.7.1 and above.

  NOTE: you have to supply an opening mode that actually permits reading from the file handle.

  NOTE: The `utf8` and `open_mode` options cannot be used at the same time.

Debugging Options

- debug

  If set to 1 the module will write random debugging information to STDERR. Defaults to 0.

- stack_debug

  If set to 1 the module will use Data::Dumper to print out the contents of the parse_stack to STDERR. Defaults to 0.

- cache_debug

  If set to 1 the module will send information on cache loads, hits and misses to STDERR. Defaults to 0.

- shared_cache_debug

  If set to 1 the module will turn on the debug option in IPC::SharedCache. Defaults to 0.

- memory_debug

  If set to 1 the module will send information on cache memory usage to STDERR. Requires the GTop module. Defaults to 0.

Miscellaneous Options

- associate

  This option allows you to inherit the parameter values from other objects. The only requirement for the other object is that it have a `param()` method that works like HTML::Template's `param()`. A good candidate would be a CGI query object. Example:

  ```
  my $query    = CGI->new;
  my $template = HTML::Template->new(
      filename  => 'template.tmpl',
      associate => $query,
  );
  ```

  Now, `$template->output()` will act as though

  ```
  $template->param(form_field => $cgi->param('form_field'));
  ```

  had been specified for each key/value pair that would be provided by the `$cgi->param()` method. Parameters you set directly take precedence over associated parameters.

  You can specify multiple objects to associate by passing an anonymous array to the associate option. They are searched for parameters in the order they appear:

```
my $template = HTML::Template->new(
    filename  => 'template.tmpl',
    associate => [$query, $other_obj],
);
```

NOTE: The parameter names are matched in a case-insensitive manner. If you have two parameters in a CGI object like 'NAME' and 'Name' one will be chosen randomly by associate. This behavior can be changed by the `case_sensitive` option.

- case_sensitive

  Setting this option to true causes HTML::Template to treat template variable names case-sensitively. The following example would only set one parameter without the `case_sensitive` option:

  ```
  my $template = HTML::Template->new(
      filename       => 'template.tmpl',
      case_sensitive => 1
  );
  $template->param(
      FieldA => 'foo',
      fIELDa => 'bar',
  );
  ```

  This option defaults to off.

  NOTE: with `case_sensitive` and `loop_context_vars` the special loop variables are available in lower-case only.

- loop_context_vars

  When this parameter is set to true (it is false by default) extra variables that depend on the loop's context are made available inside a loop. These are:

  - __first__

    Value that is true for the first iteration of the loop and false every other time.

  - __last__

    Value that is true for the last iteration of the loop and false every other time.

  - __inner__

    Value that is true for the every iteration of the loop except for the first and last.

  - __outer__

    Value that is true for the first and last iterations of the loop.

  - __odd__

    Value that is true for the every odd iteration of the loop.

  - __even__

    Value that is true for the every even iteration of the loop.

  - __counter__

    An integer (starting from 1) whose value increments for each iteration of the loop.

  - __index__

    An integer (starting from 0) whose value increments for each iteration of the loop.

  Just like any other `TMPL_VAR`s these variables can be used in `<TMPL_IF>`, `<TMPL_UNLESS>` and `<TMPL_ELSE>` to control how a loop is output.

Example:

```
    <TMPL_LOOP NAME="FOO">
      <TMPL_IF NAME="__first__">
        This only outputs on the first pass.
      </TMPL_IF>

      <TMPL_IF NAME="__odd__">
        This outputs every other pass, on the odd passes.
      </TMPL_IF>

      <TMPL_UNLESS NAME="__odd__">
        This outputs every other pass, on the even passes.
      </TMPL_UNLESS>

      <TMPL_IF NAME="__inner__">
        This outputs on passes that are neither first nor last.
      </TMPL_IF>

      This is pass number <TMPL_VAR NAME="__counter__">.

      <TMPL_IF NAME="__last__">
        This only outputs on the last pass.
      </TMPL_IF>
    </TMPL_LOOP>
```

One use of this feature is to provide a "separator" similar in effect to the perl function `join()`. Example:

```
    <TMPL_LOOP FRUIT>
      <TMPL_IF __last__> and </TMPL_IF>
      <TMPL_VAR KIND><TMPL_UNLESS __last__>, <TMPL_ELSE>.</TMPL_UNLESS>
    </TMPL_LOOP>
```

Would output something like:

```
  Apples, Oranges, Brains, Toes, and Kiwi.
```

Given an appropriate `param()` call, of course. **NOTE**: A loop with only a single pass will get both `__first__` and `__last__` set to true, but not `__inner__`.

- no_includes

  Set this option to 1 to disallow the `<TMPL_INCLUDE>` tag in the template file. This can be used to make opening untrusted templates **slightly** less dangerous. Defaults to 0.

- max_includes

  Set this variable to determine the maximum depth that includes can reach. Set to 10 by default. Including files to a depth greater than this value causes an error message to be displayed. Set to 0 to disable this protection.

- die_on_missing_include

  If true, then HTML::Template will die if it can't find a file for a `<TMPL_INCLUDE>`. This defaults to true.

- global_vars

  Normally variables declared outside a loop are not available inside a loop. This option makes `<TMPL_VAR>`s like global variables in Perl − they have unlimited scope. This option also affects `<TMPL_IF>` and `<TMPL_UNLESS>`.

Example:

```
This is a normal variable: <TMPL_VAR NORMAL>.<P>

<TMPL_LOOP NAME=FROOT_LOOP>
  Here it is inside the loop: <TMPL_VAR NORMAL><P>
</TMPL_LOOP>
```

Normally this wouldn't work as expected, since `<TMPL_VAR NORMAL>`'s value outside the loop is not available inside the loop.

The global_vars option also allows you to access the values of an enclosing loop within an inner loop. For example, in this loop the inner loop will have access to the value of `OUTER_VAR` in the correct iteration:

```
<TMPL_LOOP OUTER_LOOP>
  OUTER: <TMPL_VAR OUTER_VAR>
    <TMPL_LOOP INNER_LOOP>
       INNER: <TMPL_VAR INNER_VAR>
       INSIDE OUT: <TMPL_VAR OUTER_VAR>
    </TMPL_LOOP>
</TMPL_LOOP>
```

One side-effect of `global_vars` is that variables you set with `param()` that might otherwise be ignored when `die_on_bad_params` is off will stick around. This is necessary to allow inner loops to access values set for outer loops that don't directly use the value.

NOTE: `global_vars` is not `global_loops` (which does not exist). That means that loops you declare at one scope are not available inside other loops even when `global_vars` is on.

- filter

This option allows you to specify a filter for your template files. A filter is a subroutine that will be called after HTML::Template reads your template file but before it starts parsing template tags.

In the most simple usage, you simply assign a code reference to the filter parameter. This subroutine will receive a single argument − a reference to a string containing the template file text. Here is an example that accepts templates with tags that look like `!!!ZAP_VAR FOO!!!` and transforms them into HTML::Template tags:

```
my $filter = sub {
    my $text_ref = shift;
    $$text_ref =~ s/!!!ZAP_(.*?)!!!/<TMPL_$1>/g;
};

# open zap.tmpl using the above filter
my $template = HTML::Template->new(
    filename => 'zap.tmpl',
    filter   => $filter,
);
```

More complicated usages are possible. You can request that your filter receives the template text as an array of lines rather than as a single scalar. To do that you need to specify your filter using a hash-ref. In this form you specify the filter using the `sub` key and the desired argument format using the `format` key. The available formats are `scalar` and `array`. Using the `array` format will incur a performance penalty but may be more convenient in some situations.

```
my $template = HTML::Template->new(
    filename => 'zap.tmpl',
    filter   => {
        sub    => $filter,
        format => 'array',
    }
);
```

You may also have multiple filters. This allows simple filters to be combined for more elaborate functionality. To do this you specify an array of filters. The filters are applied in the order they are specified.

```
my $template = HTML::Template->new(
    filename => 'zap.tmpl',
    filter   => [
        {
            sub    => \&decompress,
            format => 'scalar',
        },
        {
            sub    => \&remove_spaces,
            format => 'array',
        },
    ]
);
```

The specified filters will be called for any `TMPL_INCLUDE`ed files just as they are for the main template file.

- default_escape

  Set this parameter to a valid escape type (see the `escape` option) and HTML::Template will apply the specified escaping to all variables unless they declare a different escape in the template.

config

A package method that is used to set/get the global default configuration options. For instance, if you want to set the `utf8` flag to always be on for every template loaded by this process you would do:

```
HTML::Template->config(utf8 => 1);
```

Or if you wanted to check if the `utf8` flag was on or not, you could do:

```
my %config = HTML::Template->config;
if( $config{utf8} ) {
    ...
}
```

Any configuration options that are valid for `new()` are acceptable to be passed to this method.

param

`param()` can be called in a number of ways

1 – To return a list of parameters in the template :

```
my @parameter_names = $self->param();
```

2 – To return the value set to a param :

```
my $value = $self->param('PARAM');
```

3 – To set the value of a parameter :

```
# For simple TMPL_VARs:
$self->param(PARAM => 'value');
```

```
        # with a subroutine reference that gets called to get the value
        # of the scalar.  The sub will receive the template object as a
        # parameter.
        $self->param(PARAM => sub { return 'value' });

        # And TMPL_LOOPs:
        $self->param(LOOP_PARAM => [{PARAM => VALUE_FOR_FIRST_PASS}, {PARAM => VAL
```

4 – To set the value of a number of parameters :

```
        # For simple TMPL_VARs:
        $self->param(
            PARAM  => 'value',
            PARAM2 => 'value'
        );

        # And with some TMPL_LOOPs:
        $self->param(
            PARAM              => 'value',
            PARAM2             => 'value',
            LOOP_PARAM         => [{PARAM => VALUE_FOR_FIRST_PASS}, {PARAM => VALU
            ANOTHER_LOOP_PARAM => [{PARAM => VALUE_FOR_FIRST_PASS}, {PARAM => VALU
        );
```

5 – To set the value of a number of parameters using a hash-ref :

```
        $self->param(
            {
                PARAM              => 'value',
                PARAM2             => 'value',
                LOOP_PARAM         => [{PARAM => VALUE_FOR_FIRST_PASS}, {PARAM =>
                ANOTHER_LOOP_PARAM => [{PARAM => VALUE_FOR_FIRST_PASS}, {PARAM =>
            }
        );
```

An error occurs if you try to set a value that is tainted if the `force_untaint` option is set.

clear_params

Sets all the parameters to undef. Useful internally, if nowhere else!

output

`output()` returns the final result of the template. In most situations you'll want to print this, like:

```
    print $template->output();
```

When output is called each occurrence of `<TMPL_VAR NAME=name>` is replaced with the value assigned to "name" via `param()`. If a named parameter is unset it is simply replaced with ''. `<TMPL_LOOP>`s are evaluated once per parameter set, accumulating output on each pass.

Calling `output()` is guaranteed not to change the state of the HTML::Template object, in case you were wondering. This property is mostly important for the internal implementation of loops.

You may optionally supply a filehandle to print to automatically as the template is generated. This may improve performance and lower memory consumption. Example:

```
    $template->output(print_to => *STDOUT);
```

The return value is undefined when using the `print_to` option.

query

This method allow you to get information about the template structure. It can be called in a number of ways. The simplest usage of query is simply to check whether a parameter name exists in the template, using the `name` option:

```
if ($template->query(name => 'foo')) {
    # do something if a variable of any type named FOO is in the template
}
```

This same usage returns the type of the parameter. The type is the same as the tag minus the leading 'TMPL_'. So, for example, a `TMPL_VAR` parameter returns 'VAR' from `query()`.

```
if ($template->query(name => 'foo') eq 'VAR') {
    # do something if FOO exists and is a TMPL_VAR
}
```

Note that the variables associated with `TMPL_IF`s and `TMPL_UNLESS`s will be identified as 'VAR' unless they are also used in a `TMPL_LOOP`, in which case they will return 'LOOP'.

`query()` also allows you to get a list of parameters inside a loop (and inside loops inside loops). Example loop:

```
<TMPL_LOOP NAME="EXAMPLE_LOOP">
  <TMPL_VAR NAME="BEE">
  <TMPL_VAR NAME="BOP">
  <TMPL_LOOP NAME="EXAMPLE_INNER_LOOP">
    <TMPL_VAR NAME="INNER_BEE">
    <TMPL_VAR NAME="INNER_BOP">
  </TMPL_LOOP>
</TMPL_LOOP>
```

And some query calls:

```
# returns 'LOOP'
$type = $template->query(name => 'EXAMPLE_LOOP');

# returns ('bop', 'bee', 'example_inner_loop')
@param_names = $template->query(loop => 'EXAMPLE_LOOP');

# both return 'VAR'
$type = $template->query(name => ['EXAMPLE_LOOP', 'BEE']);
$type = $template->query(name => ['EXAMPLE_LOOP', 'BOP']);

# and this one returns 'LOOP'
$type = $template->query(name => ['EXAMPLE_LOOP', 'EXAMPLE_INNER_LOOP']);

# and finally, this returns ('inner_bee', 'inner_bop')
@inner_param_names = $template->query(loop => ['EXAMPLE_LOOP', 'EXAMPLE_INNER

# for non existent parameter names you get undef this returns undef.
$type = $template->query(name => 'DWEAZLE_ZAPPA');

# calling loop on a non-loop parameter name will cause an error. This dies:
$type = $template->query(loop => 'DWEAZLE_ZAPPA');
```

As you can see above the `loop` option returns a list of parameter names and both `name` and `loop` take array refs in order to refer to parameters inside loops. It is an error to use `loop` with a parameter that is not a loop.

Note that all the names are returned in lowercase and the types are uppercase.

Just like `param()`, `query()` with no arguments returns all the parameter names in the template at the top level.

LAZY VALUES

As mentioned above, both `TMPL_VAR` and `TMPL_LOOP` values can be code references. These code references are only executed if the variable or loop is used in the template. This is extremely useful if you want to make a variable available to template designers but it can be expensive to calculate, so you only want to do so if you have to.

Maybe an example will help to illustrate. Let's say you have a template like this:

```
<tmpl_if we_care>
  <tmpl_if life_universe_and_everything>
</tmpl_if>
```

If `life_universe_and_everything` is expensive to calculate we can wrap it's calculation in a code reference and HTML::Template will only execute that code if `we_care` is also true.

```
$tmpl->param(life_universe_and_everything => sub { calculate_42() });
```

Your code reference will be given a single argument, the HTML::Template object in use. In the above example, if we wanted `calculate_42()` to have this object we'd do something like this:

```
$tmpl->param(life_universe_and_everything => sub { calculate_42(shift) });
```

This same approach can be used for `TMPL_LOOP`s too:

```
<tmpl_if we_care>
  <tmpl_loop needles_in_haystack>
    Found <tmpl_var __counter>!
  </tmpl_loop>
</tmpl_if>
```

And in your Perl code:

```
$tmpl->param(needles_in_haystack => sub { find_needles() });
```

The only difference in the `TMPL_LOOP` case is that the subroutine needs to return a reference to an ARRAY, not just a scalar value.

Multiple Calls

It's important to recognize that while this feature is designed to save processing time when things aren't needed, if you're not careful it can actually increase the number of times you perform your calculation. HTML::Template calls your code reference each time it seems your loop in the template, this includes the times that you might use the loop in a conditional (`TMPL_IF` or `TMPL_UNLESS`). For instance:

```
<tmpl_if we care>
  <tmpl_if needles_in_haystack>
      <tmpl_loop needles_in_haystack>
        Found <tmpl_var __counter>!
      </tmpl_loop>
  <tmpl_else>
    No needles found!
  </tmpl_if>
</tmpl_if>
```

This will actually call `find_needles()` twice which will be even worse than you had before. One way to work around this is to cache the return value yourself:

```
my $needles;
$tmpl->param(needles_in_haystack => sub { defined $needles ? $needles : $need
```

BUGS

I am aware of no bugs − if you find one, join the mailing list and tell us about it. You can join the HTML::Template mailing-list by visiting:

```
http://lists.sourceforge.net/lists/listinfo/html-template-users
```

Of course, you can still email me directly (`sam@tregar.com`) with bugs, but I reserve the right to forward bug reports to the mailing list.

When submitting bug reports, be sure to include full details, including the VERSION of the module, a test script and a test template demonstrating the problem!

If you're feeling really adventurous, HTML::Template has a publically available Git repository. See below for more information in the PUBLIC GIT REPOSITORY section.

CREDITS

This module was the brain child of my boss, Jesse Erlbaum (`jesse@vm.com`) at Vanguard Media (http://vm.com) . The most original idea in this module – the `<TMPL_LOOP>` – was entirely his.

Fixes, Bug Reports, Optimizations and Ideas have been generously provided by:

- Richard Chen
- Mike Blazer
- Adriano Nagelschmidt Rodrigues
- Andrej Mikus
- Ilya Obshadko
- Kevin Puetz
- Steve Reppucci
- Richard Dice
- Tom Hukins
- Eric Zylberstejn
- David Glasser
- Peter Marelas
- James William Carlson
- Frank D. Cringle
- Winfried Koenig
- Matthew Wickline
- Doug Steinwand
- Drew Taylor
- Tobias Brox
- Michael Lloyd
- Simran Gambhir
- Chris Houser <chouser@bluweb.com>
- Larry Moore
- Todd Larason
- Jody Biggs
- T.J. Mather
- Martin Schroth
- Dave Wolfe
- uchum
- Kawai Takanori

- Peter Guelich
- Chris Nokleberg
- Ralph Corderoy
- William Ward
- Ade Olonoh
- Mark Stosberg
- Lance Thomas
- Roland Giersig
- Jere Julian
- Peter Leonard
- Kenny Smith
- Sean P. Scanlon
- Martin Pfeffer
- David Ferrance
- Gyepi Sam
- Darren Chamberlain
- Paul Baker
- Gabor Szabo
- Craig Manley
- Richard Fein
- The Phalanx Project
- Sven Neuhaus
- Michael Peters
- Jan Dubois
- Moritz Lenz

Thanks!

WEBSITE

You can find information about HTML::Template and other related modules at:

```
http://html-template.sourceforge.net
```

PUBLIC GIT REPOSITORY

HTML::Template now has a publicly accessible Git repository provided by GitHub (github.com). You can access it by going to https://github.com/mpeters/html−template. Give it a try!

AUTHOR

Sam Tregar, `sam@tregar.com`

CO-MAINTAINER

Michael Peters, `mpeters@plusthree.com`

LICENSE