

phemplate: User Guide

Author: [pukomuko](#)

Last modified: 2004.03.04

Table Of Contents

1.What is phemplate.....	3
2.Usage	4
2.1.Intro	5
2.1.1.Handles.....	6
2.1.2.Loops.....	7
2.1.3.Blocks.....	9
2.1.4.Optional.....	10
2.1.5.Include.....	11
2.1.6.Tied vars and tied loops.....	12
2.1.7.UTF headers.....	13
2.2.Options	13
2.2.1.Root folder.....	14
2.2.2.Default process() options.....	14
2.2.3.Unknown variables.....	14
2.2.4.Block syntax.....	14
2.2.5.Error handler.....	15
2.3.process() options	16
2.3.1.TPL_LOOP, TPL_NOLOOP.....	16
2.3.2.TPL_INCLUDE.....	16
2.3.3.TPL_APPEND.....	16
2.3.4.TPL_FINISH.....	16
2.3.5.TPL_OPTIONAL.....	17
2.3.6.TPL_OPTLOOP.....	17
3.License.....	17

1. What is phemplate

phemplate is simple and fast templating engine for [php](#). It provides a way of substituting variables into text templates and do some dynamic block functionality including loops.

Templating engines are meant for separation of presentation and logic. That means you can put all your HTML content outside of PHP scripts. HTML can be changed by designers without a fear of messing up your scripts.

There are no big reasons to choose **phemplate** and I encourage you to try [Smarty](#). Smarty tries to invent new programming language not only for presentantion, but also for presentation logic. Using phemplate you code presentation logic in PHP scripts.

2. Usage

phemplate is not limited to HTML. It can work on all text based documents. You can use it for WAP, XML, configuration files, code generation, etc. Basic workflow using phemplate looks like this:

- create phemplate object: `$tpl = new phemplate()`
- set up some text content: `$tpl->set_file('content', 'template.html')`
- set up variables: `$tpl->set_var('name', $value)`
- substitute variables in text: `$tpl->process('', 'content')`

phemplate has only two value containers: [handles](#) and [loops](#). All text, files and variables are placed inside handles container. Loops are held in special container. Another concept is [block](#). Blocks are used to split files into separate segments. You can later manipulate them: repeat, combine, choose only one depending on script logic.

2.1.Intro

2.1.1.Handles

2.1.2.Loops

2.1.3.Blocks

2.1.4.Optional

2.1.5.Include

2.1.6.Tied vars and tied loops

2.1.7.UTF headers

2.2.Options

2.2.1.Root folder

2.2.2.Default process() options

2.2.3.Unknown variables

2.2.4.Block syntax

2.2.5.Error handler

2.3.process() options

2.3.1.TPL_LOOP, TPL_NOLOOP

2.3.2.TPL_INCLUDE

2.3.3.TPL_APPEND

2.3.4.TPL_FINISH

2.3.5.TPL_OPTIONAL

2.3.6.TPL_OPTLOOP

2.1. Intro

Simple script:

```
<?
include('phemplate.class.php');
$tpl = new phemplate();
$tpl->set_var('var', 'world');
$tpl->set_file('text', '1.plain.html');
echo $tpl->process("", 'text');
?>
```

Template:

Some text.
Hello {var}!

Output:

Some text.
Hello world!

- [2.1.1.Handles](#)
- [2.1.2.Loops](#)
- [2.1.3.Blocks](#)
- [2.1.4.Optional](#)
- [2.1.5.Include](#)
- [2.1.6.Tied vars and tied loops](#)
- [2.1.7.UTF headers](#)

2.1.1. Handles

Handle is basically a text with a name. All text bits in phemplate have names. When phemplate encounters `{var}` in template it tries to substitute this string with value of handle `'var'`.

Calling `set_var('var', 'some text')` you assign value `'some text'` to handle `'var'`.

Calling `set_file('text', 'file.html')` you assign contents of file `file.html` to handle `'text'`.

You can set as variables not only scalar values but also arrays. Let's suppose you have array and pass it to `set_var()`:

```
$data = array(
    'one' => 1,
    'two' => array(
        'two1' => 21,
        'two2' => 22
    ),
    'three' => 3
);
$tpl->set_var('info', $data);
```

You can access data of this array in template by `{info.one}`, `{info.two.two1}`, `{info.two.two2}`, `{info.three}`.

2.1.2. Loops

One of the most common tasks in template usage is to iterate over some kind of array. Reports, lists, menus are constructed using this technique. phemplate has special construct for regularly structured arrays: `<loop>`

phemplate only accepts this structure as loops: array with keys [0..n] (ordered integer numbers) and values as associative arrays having the same keys.

```
$loop = array (
    '0' => array (
        'id' => 3,
        'info' => 'first row'
    ),
    '1' => array (
        'id' => 233,
        'info' => 'second row'
    ),
    '2' => array (
        'id' => 56,
        'info' => 'third row'
    )
);
```

Use `<loop>` tag to markup loop statement in template file. This tag must be lowercase, between 'loop' and 'name' must be only one space character.

Header

```
<loop name="info">
    {info.id}, {info.info}<br>
</loop name="info">
```

Script:

```
<?
include('phemplate.class.php');
$tpl = new phemplate();
$tpl->set_loop('info', $loop); // from above code
$tpl->set_file('text', 'loop.html');
echo $tpl->process('', 'text', TPL_LOOP);
?>
```

Result:

Header

3, first row

233, second row

56, third row

If loop has no elements nothing is shown. If you want phemplate to show some message in case of empty loop, use `<noloop>` tag. It has the same restrictions as `<loop>` and must be nested inside it. Also `process()` function must receive flag, to check for noloop tag:
`$tpl->process('out', 'main', TPL_NOLOOP);` It is turned off by default.

```
<loop name="info">
  {info.id}, {info.info}<br>
  <noloop name="info">
    The list is empty.
  </noloop name="info">
</loop name="info">
```

2.1.3. Blocks

Blocks are used to split files into separate segments. When reading a file you must explicitly tell phemplate to look for blocks: `set_file('handle', 'filename', TPL_BLOCK)`. Found templates are taken out from file handle and put into their own handles.

Template:

```
Some text.  
<block name="new_block">  
NEW TEXT!  
</block name="new_block">  
Other text.
```

Simple script:

```
<?  
include('phemplate.class.php');  
$tpl = new phemplate();  
$tpl->set_file('text', '2.block.html', TPL_BLOCK);  
echo $tpl->process("", 'text');  
echo $tpl->process("", 'new_block');  
>
```

Output:

Some text.

Other text.
NEW TEXT!

Note: you can define blocks inside blocks, but `set_file()` must use `TPL_BLOCKREC`

2.1.4. Optional

Text between `<opt name="var">` tags is shown only if `var` is set and has not false value (`var!='', var!=false, var!=null`).

Note: values `0` and `'0'` are considered not false.

You can put variables inside that text, but loops won't be parsed. You need second pass for loops.

info

```
<opt name="var">some more text {var} </opt name="var">
<opt name="var2">this text won't be shown</opt name="var2">
```

```
$tpl->set_var('var', 'for you');
echo $tpl->process('', 'main', TPL_OPTIONAL);
```

2.1.5. Include

Sometimes it is convenient to include parts of template from other files. phemplate supports this kind of activity by `<include filename="text.html">` tag. When processing such template you must explicitly tell phemplate to look for includes: `process('dest', 'source', TPL_INCLUDE)`. I strongly recommend you to use blocks instead of includes.

Note: if you include php script it won't be executed but its source will be inserted into template.

2.1.6. Tied vars and tied loops

Calling `set_var()` and `set_loop()` data is copied inside phemplate. In general it is faster and more convenient, but sometimes it is more practical just to tell phemplate to use only reference to variable or loop. To do so you must call `tie_var()` or `tie_loop()`.

```
$var = 1;
$this->tpl->tie_var('temp', $var);
$var = 2;
echo $this->tpl->get_var('temp'); // result - 2
```

However if you tie array you cannot change its structure later only the values. It doesn't concern loops.

Note: changes are noticeable only before `process()`.

2.1.7. UTF headers

If you use UTF encoding and your text editor puts UTF byte order headers at the beginning of the file you might have a problem. You can use phemplate to combine one big page from several disparate files and it will contain these UTF headers at inappropriate places. So you can tell phemplate to strip this UTF header when reading the file using `TPL_STRIP_UTF_HEADER` option for `set_file()`.

```
$tpl->set_file('temp', 'filename.html', TPL_STRIP_UTF_HEADER);  
$tpl->set_file('temp', 'filename.html', TPL_STRIP_UTF_HEADER | TPL_BLOCK);
```

Note: this will strip the first 3 bytes if they are equal to `\xEF\xBB\xBF`.

2.2. Options

You can customize phemplate behaviour to some extent. All available options are listed below.

- [2.2.1. Root folder](#)
- [2.2.2. Default process\(\) options](#)
- [2.2.3. Unknown variables](#)
- [2.2.4. Block syntax](#)
- [2.2.5. Error handler](#)

2.2.1. Root folder

You can place all your template files in separate folder and use `set_root($folder_name)`. After that all calls to `set_file('handle', $file)` will be translated to `set_file('handle', $folder_name.$file)`. Root folder is also the first parameter in pemplate constructor `pemplate($folder_name)`.

Note: ending slash is required for folder name: `module/templates/`

Tip: you can set root folder to working folder: `set_root('./')`

2.2.2. Default process() options

`process()` has many available options. You can set default set of options in constructor or calling `set_params()`. Third parameter in constructor specifies default options for `process():pemplate($root, $unknown, $params)`.

```
include('pemplate.class.php');
$tpl = new pemplate('./', 'keep', TPL_NOLOOP | TPL_FINISH);
$tpl->set_params(TPL_NOLOOP | TPL_FINISH | TPL_OPTIONAL);
```

Note: default options are only used when `process()` is called without any options.

2.2.3. Unknown variables

Unknown variable results when you have `{var}` in template, but handle 'var' is not set. Second parameter in constructor specifies what to do with unknown variables `pemplate($root, $unknown)`. You can also set this parameter by calling method `set_unknowns($what)`.

Possible values for this parameter:

- **keep** - default value, all unknown variables are kept intact in text.
- **remove** - remove all unknown variables.
- **remove_nonjs** - remove only variables that have no space, `\t` or `\n` inside. Good for preserving JavaScript or CSS in HTML files.
- **comment** - `<!-- {var} -->`
- **space** - change variable to ` `;

Note: as for v1.9 `TPL_FINISH` does not support `remove_nonjs`. It's only in normal `process()` phase.

2.2.4. Block syntax

You can change the way blocks are described in template file. By default it is:

```
content
  <block name="handle">block content</block name="handle">
content
```

But with `set_block_syntax($start, $end)` you can change this. Example:
`set_block_syntax('<!-- BLOCK START | -->', '<!-- BLOCK END | -->')`.
Now the above block must look like this:

```
content
  <!-- BLOCK START handle -->block content<!-- BLOCK END handle -->
content
```

There are some restrictions: start and end tags must differ, they must contain symbol '|' which is placeholder for handle name, tags are case sensitive.

2.2.5. Error handler

phemplate reports encountered errors and warnings. By default these are printed to browser. You can set custom error handler for catching errors and warnings from phemplate `set_error_handler($handler)`. Error handler `$handler` must be object with method `report($level, $msg)`. `$msg` contains warning or error message. `$level` is `E_USER_WARNING` for warnings and `E_USER_ERROR` for errors.

Simple error handler:

```
class errorhandler
{
  function report($lvl, $msg)
  {
    print $msg;
    if (E_USER_ERROR == $lvl) die('fatal phemplate error');
  }
}
```

Note: error handler must not return control to phemplate after error.

2.3. process() options

`process()` is the method that does all the work of phemplate. The syntax is this:
`process($destination, $source, $params)`. `$source` and `$destination` are names of handles. Text of `$source` is taken and transformed according to `$params`. The result is put into handle `$destination` and also returned as result of `process()`.

`$params` can contain more than one option. This is achieved by using bitwise or operator `|`.

```
$tpl->process('dest', 'source', TPL_NOLOOP | TPL_FINISH | TPL_OPTIONAL);  
$tpl->process('dest', 'source', TPL_OPTIONAL);  
$tpl->process('dest', 'source', TPL_NOLOOP | TPL_FINISH | TPL_OPTIONAL | TPL_APPEND);
```

[2.3.1.TPL_LOOP, TPL_NOLOOP](#)

[2.3.2.TPL_INCLUDE](#)

[2.3.3.TPL_APPEND](#)

[2.3.4.TPL_FINISH](#)

[2.3.5.TPL_OPTIONAL](#)

[2.3.6.TPL_OPTLOOP](#)

2.3.1. TPL_LOOP, TPL_NOLOOP

`TPL_LOOP` and `TPL_NOLOOP` options tells phemplate to look for `<loop>` and `<no loop>` tags respectively.

2.3.2. TPL_INCLUDE

`TPL_INCLUDE` option tells phemplate to look for `<include filename="somefile">` tags.

2.3.3. TPL_APPEND

`TPL_APPEND` option tells phemplate not to replace `$destination` with result, but to append result to the end of `$destination`. If no `$destination` exists empty one will be created first.

2.3.4. TPL_FINISH

TPL_FINISH option tells phemplate to look for {vars} left in phemplate and remove, comment or space them depending on [unknowns](#) setting.

2.3.5. TPL_OPTIONAL

TPL_OPTIONAL option tells phemplate to look for `<opt name="somevar">` tags.

2.3.6. TPL_OPTLOOP

TPL_OPTLOOP option tells phemplate to look for `<opt name="loopname.somevar">` tags inside loops.

3. License

this software is provided 'as-is', without any express or implied warranty. in no event will the authors be held liable for any damages arising from the use of this software.

permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. the origin of this software must not be misrepresented; you must not claim that you wrote the original software. if you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. mail about the fact of using this class in production would be very appreciated.
4. this notice may not be removed or altered from any source distribution.

