Template Engine

Clay Dowling
Lazarus Internet Development

November 18, 2004

Contents

1 Overview

The template engine allows you to separate the logic of your CGI program from the display of it. For the developer, this means that the designers and suits can fuss about the appearance of the application until the cows come home and it won't affect your application. For the designers, it means that your elegant design doesn't get mangled by some oaf of a developer. And for the suits, it means you can put your stamp on an application without having to blow deadlines and budgets on overtime for expensive developers.

The Template Engine is a collection of smaller templates. A template consists of literal text and *elements*. An element in a template will be replaced by a value set in the template engine.

When a template is parsed, the elements are replaced with their set values, and the parsed template is stored in an new element. The developer has the option of replacing any existing value for the template, or appending to the existing value.

Applications deal with element values as a `char*`, making it easy to use their values in your application.

A single web page will typically be composed of several templates. Smaller templates are parsed to become elements in larger templates.

2 Template Files

A template file can contain multiple templates. It is often convenient to put related templates in the same file.

A template in a file is enclosed in `<template>` and `</template>` tags. It has a mandatory `name` attribute. Owing to the cheap XML parser,[1] enclosing the attribute is quotes is mandatory.

Within a template, there is literal text, which will be unchanged by the parsing process, and elements. Parsing will replace the element with the values assigned in the template engine. Elements without assigned values will not appear in the parsed result.

An element is enclosed in curly braces, and may contain letters, numbers and punctuation marks (although certain marks, such as a backslash character, should be avoided due to the problems that your C compiler will give you when you try to assign a value to this element). Whitespace between curly braces will cause it to be interpretted as literal text rather than an element. This convention allows the use of javascript functions in templates.

2.1 Template File Example

A simple example template file is shown in figure 1. It consists of two templates, “row” and “grid”. The “row” template will be parsed and appended to the “rows” element in a loop. When the “grid” template is parsed, the cumulative “row” templates will appear in the “rows” element.

2.2 Operating System Compatibility

I have tried to make the library insensitive to the line-end conventions used in a template file. I am prone to editing the same file on both UNIX and Windows machines, which can cause the line-end convention to vary from point to point in a single file.

That’s no guarantee though that differences between the designer’s workstation line end conventions and the production web server’s line end conventions won’t cause trouble. For this reason, it’s a good idea to transfer your files in text mode if you are ftping your files to the server.

I do know that there are no problems using templates created on a Windows machine with a CGI application on a UNIX web server.

3 Library API

Applications using the template engine must include `template.h` and link against libtemplate. The default for the library is to build both static and dynamic libraries. If

[1] Well, not so much an XML parser as a call to `strchr`.

Figure 1: Sample Template File

```
<template name="grid">
<table>
<tr>
  <th>n</th>
  <th>n^2</th>
  <th>n^3</th>
</tr>
{rows}
</table>
</template>

<template name="row">
<tr>
  <td>{n}</td>
  <td>{n2}</td>
  <td>{n3}</td>
</tr>
</template>
```

you have little control over the administration of the web host, static libraries are a real boon. They're also useful for performance reasons.

The primary data structure is `struct tpl_engine`. For most applications, a single `tpl_engine` will be sufficient. Allocate space for and initialize the `tpl_engine` with a call to `tpl_engine_new(void)`. This will allocate space and initialize the internal data structures.

3.1 tpl_file_load

```
void tpl_file_load(struct tpl_engine* engine, char* filename)
```

This function loads a template file and populates the template engine with the templates found inside.

3.2 tpl_element_set/get

```
void tpl_element_set(struct tpl_engine* engine, char* element,
  char* value)
char* tpl_element_get(struct tpl_engine* engine, char* element)
```

Set or get the value of an individual element defined in the engine. If the value of an element has not been set, `tpl_element_get` will return NULL. You should be careful in your program to check the value of any returned element to ensure that

it is not NULL, since some string functions react poorly to a NULL as an argument.

Warning: Don't do anything rash like call `free` or `realloc` on the return value of `tpl_element_get`. This string is owned by the template engine. If the element is ever replaced, you will seriously irritate the computer. With a BSD based libc, you get a warning to stderr to the effect that the memory was already free. With a straigh GNU libc, such as Linux, mingw or Dev-C++ use, you get a seg fault. This mistake has caused a lot of errors that could have been avoided.

3.3 tpl_parse

```
void tpl_parse(struct tpl_engine* engine,
  char* templatename, char* newelement, int append)
```

This function converts your template into the named element. If you wish to append to the old value of the element, set the append argument to 1, otherwise use 0. Appending is useful for tasks like building a table from database information.

The astute programmer will note that the `template.h` header file contains many more functions and a host of data types besides those defined here. Those functions should be considered private; you will not need to understand their use in your program.

4 Example Program

The program `test.c` and the accompanyine template file `test.tpl` demonstrate how to write a simple program to build more complicated output.