C++2Any Templates Reference

Vadim Zeitlin

February 8, 2004

Contents

0 Introduction

As explained in the overview of C++2Any , it uses the template files to allow the user to customize the output of the program. This document describes the format of these template files in details.

The first chapter gives a brief overview of the template files. The syntax details are described in the chapters 2 and 3 and the appendix contains an example of a commented template file which should make it easier to understand the concepts described here. Please consult it while reading the main text.

1 Template Files Overview

The template files have, by convention, `c2a` extension, but it is not mandatory. A template is a simple text file (in 7 bit ASCII) describing the output which should be produced by the generator. The contents of the template file is normally copied to the output as is, with two exceptions which are the *variable expansions* and the *sections*.

The simplest example of variable expansions are the familiar (as they are modeled, albeit loosely, after Unix shell variables) expressions `$VAR` or `${VAR}`. Such expansions may appear in any place in the output file, inside or outside a section, and are replaced with their value on output. The paragraph 2 describes them in more details.

But the variable expansion, although a powerful mechanism by itself, is not enough as it doesn't support loops which are used by the generator all the time: for example, it may be necessary to generate some block of code for each method of a class. The sections in the template files allow to do this. A section is simply a block of text (usually containing some variable expansions) inside triple braces, e.g. `{{{<element> ... }}}`. They are described in details in the paragraph 3.

2 Variables

2.1 General Syntax

Anywhere inside the template text, a sequence of alphanumeric characters starting with the dollar sign (`'$'`) is a variable. To include a literal dollar sign in the output you should double it (`"$$"`).

A variable expansion may be either simple: `$VAR` or `${VAR}` or more complex but in this case only the latter form, with braces, can be used. If the former form is used, the first character which can't be part of the variable name, that is

anything but a letter, digit, or the underscore, ends variable name so using braces may be necessary even in a simple form, as in this example:

```
enum ${NAME}_Tag
```

as without the braces the variable name would be parsed as `NAME_Tag` which is probably not intended.

A simple variable expansion just replaces the expression in the text with the variable value. However in the `${VAR}` form, `VAR` may be a string containing other variables. For example, `${IID_$NAME}` may be replaced with the value of the variable `IID_Foo` if the variable `$NAME` has value `Foo`.

Another supported construct are Bourne shell-like expressions `${VAR+WORD}` and `${VAR-WORD}`. The value of the first of them is `WORD` if and only if variable `VAR` is set and non empty, otherwise the expansion is empty. For example, to include an optional intiializer in a variable declaration macro you could do

```
$NAME${VALUE+= $VALUE}
```

The value of the second form is `VAR` if the variable is set and not empty and `WORD` otherwise. An example is choosing the base class for an interface in the IDL template file:

```
interface $NAME : I${BASENAME-Dispatch}
```

To include a closing brace inside the expansion it has to be quoted with a dollar sign (`$}`, see 2.3.2).

Sometimes it may be also useful to reverse the effect of `${VAR+WORD}` construction, i.e. to only use the alternative value if the variable is *not* set. To achieve this you can use `${VAR!WORD}` form. For example, this fragment

```
    HRESULT $NAME({{{ PARAM* {+, }
$TYPE $NAME}}}${RETVAL+${ISVOID!, } $RETTYPE $RETVAL});}}}
```

only adds a comma (presumably used to separate the following text from the eventual parameters) if the function has any parameters.

One of the conditional forms above must be used for the variables which may not be set because an attempt to use an unset variable in any other way (for example by trying to substitute its value directly using `$NO_SUCH_VAR`) would result in a run-time error.

2.2 Variable Names

The names of the existing variables are defined by the generator being used. By convention, they are always in upper case, lower and mixed case variables are reserved for future extensions and the underscore character has a special meaning as explained below.

Variables value is defined by the section in which they appear. For example, in this fragment

```
{{{ CLASS
    class $NAME
    {
        {{{ METHOD
        void $NAME();
        }}}
    };
}}}
```

the first occruence of the variable `$NAME` stands for a class name while the second one – for the method name. To access the variable from the enclosing section, a double underscore may be used:

```
{{{ CLASS
    {{{ METHOD
        void $NAME()
        {
            puts("In $CLASS__NAME::$NAME");
        }
    }}}
}}}
```

It works as the scope resolution operator (`::`) in C++ and, accordingly, it is also valid to use `$__NAME` to refer to a global variable and not the value for the current section. On the other hand, using a `$SECT_NAME` outside of the section `SECT` is an error and is not permitted.

2.3 Built-in Variables

2.3.1 Dollar sign $$

This pseudo-variable expands to the dollar sign character. It must be used instead of bare `'$'` in the templates.

2.3.2 Closing brace `$}`

This avriable simply expands to the closing brace character. It is useful for the situations in which a bare closing brace has a special meaning (inside a conditional variable expansion or after two other closing braces, for example).

2.3.3 Comment `$/`

This variable introduces a single-line comment in the template file. Everything following `$/` until the end of line is completely ignored by 2a.

3 Sections

3.1 Syntax

A section definition starts with a sequence of three braces `"{{{"` and ends with a line consisting solely of three closing braces `"}}}"`, not counting whitespace. After the starting sequence the section name should follow, possibly after some amount of whitespace characters. The rest of the block, up to the closing sequence, is the section body and may contain arbitrary plain text, expansions of the variables (see 2) and other embedded sections.

The `{{{` and `}}}` sequences currently can't be changed nor escaped and so their special meaning can't be changed. In an unlikely event that you need to include `}}}` in the actually generated text, you must use the closing brace (see 2.3.2) variable:

```
}}$}
```

3.2 Section Names

The section name is not arbitrary. It specifies when the contents of this section is going to be generated. As with the variable names, the available section names depend on the generator used and, again, as with the variables, all standard section names are in upper case only. Note that section names don't have to be unique, it is perfectly valid to have two or more occurences of the same name.

When the parser encounters a section named `ELEMENT`, it generates the section body – expanding any variables and sections it contains in the process – for each and every element of the given type known to the generator. For example, a section named `CLASS` will be generated once for every class found in the input file.

Section names respect the scope just as the variable names do. Thus, a section named `METHOD` inside a `CLASS` section really means `CLASS_METHOD` and will be expanded for all methods of the current class.

3.3 Section Quantifiers

By default, all sections must appear at least once in the output and may do so several times (such default behaviour helps to detect accidentally misspellt section names). In some cases it may be desirable to have sections which might not appear at all (e.g. `ENUM` section is not mandatory in the output and doesn't appear if there are no enum declarations in the source file) or have those those which can't appear more than once.

To specify this in the template, one of the following characters may be appended to the section name (without intervening whitespace):

+ Specifies that the section may appear one or more times (this is the default)

? Specifies that a section may appear at most once, i.e. not appear at all or appear once.

! Signals a unique section: it must appear exactly once.

***** The most relaxed quantifier: the section may appear any number of times, including 0.

3.4 Section Parameters

After the section name (and, possibly, a quantifier) only the section parameters may appear until the end of line. All parameters are optional and they all have the form `{X...}` where `'X'` is a special character which specifies the paramater and the rest of text, until the closing brace, is the parameter value.

The following parameters are currently recognized:

+ String specified as value is used between the successive occurences of this section. By default, the string is empty for single-line sections and is a blank line for the multi-line ones, but using `{+,}`, for example, you may insert commas between sections. This parameter can only be used with sections which may appear multiple times.

- Alternative value parameter: if this section doesn't occur at all (e.g. `PARAM` section for a function not taking any parameters), the value of this parameter is substituted instead. By default it is empty and so nothing is substituted. This parameter can only be used with the optional sections.

A Appendix A: Standard Variables and Sections

This appendix describes all standard C++2Any variables and sections.

A.1 Standard Sections

ENUM Expanded for each enum declaration occuring in the source file. At global scope, only namespace-scoped enums are considered, inside a CLASS section – only the enums nested in the current class. The expansion of this section may be empty if there are no exported enums in the source file.

ENTRY Only valid inside an ENUM and expands to all enumeration values there.

CLASS Expanded for each class or struct occuring in the source file. Inside another CLASS section, only nested classes are considered, otherwise – only those at namespace scope.

METHOD Only valid inside a CLASS section and is expanded for each class method.

PARAMS Only valid inside a METHOD section and is expanded for each parameter of the generated wrapper method. The expansion of this section may be empty for void methods.

PARAMS_IMPL This is similar to PARAMS but expands into the parameters of the original ("implementation") method, not of the generated one. For the existing backends, the two are different for the methods with non-void return value as PARAMS includes the extra parameter used for the return value and PARAMS_IMPL does not.

A.2 Standard Variables

NAME The name of the containing section or the global project name outside of any section. This variable applies to all the sections.

HELP The help string, or description, extracted from the comments in the source file. It may be empty.

VALUE Only valid inside ENTRY section and stands for the value of the enum entry. It may be empty.

BASE Only valid inside CLASS section and stands for the base class of the current class. It is empty if the class doesn't have any base classes.

TYPE Only valid inside PARAM section and stands for the parameter type there.

B Appendix B: Example of a Template File

This appendix contains a full example of a template file:

```
// Generated by cpp2any at $DATE
//
// Don’t edit directly, your changes will be lost!

import "unknwn.idl";
import "oaidl.idl";

{{{ ENUM*
[ helpstring("$HELP") ]
typedef enum ${NAME}_Tag
{
    {{{ ENTRY
    [ helpstring("$HELP") ]
    $NAME${VALUE+= $VALUE}$,
    }}}
} $NAME;
}}}

// forward declare all interfaces
{{{ CLASS
interface I$NAME;
}}}


{{{ CLASS
[
    object,
    dual,
    uuid(${IID_$NAME}), // IID_$NAME
    helpstring("$HELP")
]
interface I$NAME : ${BASE-IDispatch}
{
{{{ METHOD
    [ helpstring("$HELP") ]
    HRESULT $NAME({{{ PARAM* {+, } {0void}
                  [ $INOUT ] $TYPE $NAME}}});
```

```
}}}
};
}}}

[
    uuid(${LIBID_$NAME}),     // LIBID_$NAME
    helpstring("$HELP"),
    version(1.0)
]
library $NAME
{
    importlib("stdole2.tlb");

    {{{ CLASS
    [
        uuid(${CLSID_$NAME}),  // CLSID_$NAME
        helpstring("$HELP")
    ]
    coclass $NAME
    {
        [ default ] interface I$NAME;
    };
    }}}
};
```