

Documentation System

5

Introduction

While SWIG makes it easy to build interfaces, it is often difficult to keep track of all of the different functions, variables, constants, and other objects that have been wrapped. This especially becomes a problem when your interface starts to grow in size from a handful to several hundred functions. To address these concerns, SWIG can automatically generate documentation in a number of formats including ASCII, HTML, and LaTeX. The goal is that you could look at the documentation file to see what functions were wrapped and how they are used in the target scripting language.

Usage documentation is generated for each declaration found in an interface file. This documentation is generated by the target language module so the Tcl module will follow Tcl syntax, the Perl module will use Perl syntax, and so on. In addition, C/C++ comments can be used to add descriptive text to each function. Comments can be processed in a number of different styles to suit personal preferences or to match the style used in a particular input file.

Automatic documentation generation for C/C++ programs is a fairly formidable problem and SWIG was never intended to be a substitute for a full-blown documentation generator. However, I feel that it does a reasonable job of documenting scripting language interfaces. It seems to do just fine for many of SWIG's primary applications--rapid prototyping, debugging, and development.

How it works

For each declaration in an interface file, SWIG creates a "Documentation Entry." This entry contains three components; (1) a usage string, (2) a C information string, and (3) descriptive text. For example, suppose you have this declaration in an interface file :

```
int fact(int n);
/* This function computes a factorial */
```

The documentation entry produced by the SWIG ASCII module will look like this for Tcl:

```
fact n
  [ returns int ]
  This function computes a factorial
```

The first line shows how to call the function, the second line shows some additional information about the function (related to its C implementation), while the third line contains the comment

text. The first two lines are automatically generated by SWIG and may be different for each language module. For example, the Perl5 module would generate the following output :

```
fact($n)
  [ returns int ]
  This function computes a factorial
```

Of course, this is only a simple example, more sophisticated things are possible.

Choosing a documentation format

The type of documentation is selected using the following command line options :

-dascii	Produce ASCII documentation
-dhtml	Produce HTML documentation
-dlatex	Produce LaTeX documentation
-dnone	Produce no documentation

The various documentation modules are implemented in a manner similar to language modules so the exact choice may change in the future. With a little C++ hacking, it is also possible for you to add your own modules to SWIG. For example, with a bit of work you could turn all of the documentation into an online help command in your scripting language.

Function usage and argument names

The function usage string is produced to match the declaration given in the SWIG interface file. The names of arguments can be specified by using argument names. For example, the declarations

```
void insert_item(List *, char *);
char *lookup_item(char *name);
```

will produce the following documentation (for Python) :

```
insert_item(List *, char *)
  [ returns void ]

lookup_item(name)
  [ returns char * ]
```

When argument names are omitted, SWIG will use the C datatypes of the arguments in the documentation. If an argument name is specified, SWIG will use that in the documentation instead. Of course, it is up to each language module to create an appropriate usage string so your results may vary depending on how things have been implemented in each module.

Titles, sections, and subsections

The SWIG documentation system is hierarchical in nature and is organized into a collection of sections, subsections, subsections, and so on. The following SWIG directives can be used to organize an interface file :

- `%title` "Title Text". Set the documentation title (may only be used once)
- `%section` "Section title". Start a new section.
- `%subsection` "Subsection title". Create a new subsection.
- `%subsubsection` "Subsubsection title". Create a new subsubsection.

The `%title` directive should be placed prior to the first declaration in an interface file and may only be used once (subsequent occurrences will simply be ignored). The section directives may be placed anywhere. However, `%subsection` can only be used after a `%section` directive and `%subsubsection` can only be used after a `%subsection` directive.

With the organization directives, a SWIG interface file looks something like this :

```
%title "Example Interface File"
%module example
%{
#include "my_header.h"
%}

%section "Mathematical Functions"

... declarations ...

%section "Graphics"
%subsection "2D Plotting"
... Declarations ...
%subsection "3D Plotting"
%subsubsection "Viewing transformations"
... Declarations ...
%subsubsection "Lighting"
... Declarations ...
%subsubsection "Primitives"
... Declarations ...

%section "File I/O"

... Declarations ...
```

Formatting

Documentation text can be sorted, chopped, sliced, and diced in a variety of ways. Formatting information is specified using a comma separated list of parameters after the `%title`, `%section`, `%subsection`, or `%subsubsection` directives. For example :

```
%title "My Documentation", sort, before, pre
```

This tells SWIG to sort all of the documentation, use comments that are before each declaration, and assume that text is preformatted. These formatting directives are applied to all children in the documentation tree--in this case, everything in an interface file.

If formatting information is specified for a section like this

```
%subsection "3D Graphics", nosort, after
```

then the effect will only apply to that particular section (and all of its subsections). In this case, the formatting of the subsection would override any previous formatting, but these changes would only apply to this subsection. The next subsection could use its own formatting or that of its parent.

Style parameters can also be specified using the `%style` and `%localstyle` parameters. The `%style` directive applies a new format to the current section and all of its parents. The `%localstyle` directive applies a new format to the current section. For example :

```
%style sort,before, skip=1      # Apply these formats everywhere
%localstyle sort                # Apply this format to the current section
```

Use of these directives usually isn't required since it's easy enough to simply specify the information after each section.

Default Formatting

By default, SWIG will reformat comment text, produce documentation in the order encountered in an interface file (nosort), and annotate descriptions with a C information string. This behavior most closely matches that used in SWIG 1.0, although it is not an exact match due to differences in the old documentation system.

When used in the default mode, comment text may contain documentation specific formatting markup. For example, you could embed LaTeX or HTML markup in comments to have precise control over the look of the final document.

Comment Formatting variables

The default formatting can be changed by changing one or more of the following formatting variables :

after	Use comments after a declaration (default)
before	Use comments before a declaration
chop_top=nlines	Comment chopping (preformatted)
chop_bottom=nlines	Comment chopping (preformatted)
chop_left=nchar	Comment chopping (preformatted)
chop_right=nchar	Comment chopping (preformatted)
format	Allow SWIG to reformat text (the default)
ignore	Ignore comments
info	Print C information text (default)
keep	Keep comments (opposite of ignore)
noinfo	Don't print C information text
nosort	Don't sort documentation (default)
pre	Assume text is preformatted
skip=nlines	Number of blank lines between comment and declaration
sort	Sort documentation
tabify	Leave tabs intact
untabify	Convert tabs to spaces (default)

More variables may be available depending on particular documentation modules. The use of these variables is described in the next few sections.

The chopping parameters can be used to strip out the text of block comments. For example, using `chop_left=3`, `chop_top=1`, `chop_bottom=1` on the above comment produces the following output :

```
Plot2D_line x1 y1 x2 y2 color
  [ returns void ]
  void Plot2D_line(int x1, int y1, int x2, int y2, Pixel color)

  Draws a line between the points (x1,y1) and (x2,y2) using the
  the given color.  The line is cropped to fit in the current
  bounding box.

  Uses the Bresenham line drawing algorithm.
```

The chopping parameters only apply if a comment is sufficiently large (i.e.. if the number of lines exceed `chop_top+chop_bottom`). Thus, in our example, a one line comment will be unaltered even though chopping has been set. By default, SWIG sets `chop_left=3` and all others to zero. This setting removes the `/*` or `/**` preceding a comment.

Tabs and other annoyances

When using the preformatted mode, SWIG will automatically convert tabs to white space. This is done assuming that tabs are placed every 8 characters. The tabification mode can be selected using the `'tabify'` and `'untabify'` parameters :

```
%section "Untabified Section", untabify
%section "Leave those tabs alone", tabify
```

Tabs are simply ignored when comments are reformatted (well, actually, they're just copied into the output, but the target documentation method will ignore them).

Ignoring comments

To ignore the comments in a particular section, you can use the `'ignore'` parameter. For example :

```
%section "No Comments", ignore
%section "Keep Comments", keep
```

The `'keep'` parameter is used to disable the effect of an ignore parameter (if set by a section's parent).

C Information

Normally, each declaration in a file will have a C information tag attached to it. This is usually enclosed in `[]` and contains the return type of a function along with other information. This text can disabled using the `'noinfo'` parameters and reenabled using the `'info'` parameter.

```
%section "No C Information", noinfo
%section "Print C Information", info
```

Adding Additional Text

Additional documentation text can be added using the `%text` directive as shown :

```
%text %{  
  
This is some additional documentation text.  
  
%}
```

The `%text` directive is primarily used to add text that is not associated with any particular declaration. For example, you may want to provide a general description of a module before defining all of the functions. Any text can be placed inside the `{ , % }` block except for a `'% }` that ends the block. For the purposes of sorting, text segments will always appear immediately after the previous declaration.

Disabling all documentation

All documentation can be suppressed for a portion of an interface file by using the `%disable-doc` and `%enabledoc` directives. These would be used as follows:

```
%disabledoc  
... A bunch of declarations with no documentation ...  
%enabledoc  
... Now declarations are documented again ...
```

These directives can be safely nested. Thus, the occurrence of these directives inside a `%disable-doc` section has no effect (only the outer-most occurrence is important).

The primary use of these directives is for disabling the documentation on commonly used modules that you might use repeatedly (but don't want any documentation for). For example :

```
%disabledoc  
%include wish.i  
%include array.i  
%include timer.i  
%enabledoc
```

An Example

To illustrate the documentation system in action, here is some code from the SWIG library file 'array.i'.

```
//  
// array.i  
// This SWIG library file provides access to C arrays.  
  
%module carray  
  
%section "SWIG C Array Module", info, after, pre, nosort, skip=1, chop_left=3,  
chop_right=0, chop_top=0, chop_bottom=0  
  
%text %{  
%include array.i
```

This module provides scripting language access to various kinds of C/C++ arrays. For each datatype, a collection of four functions are created :

```
<type>_array(size)           : Create a new array of given size
<type>_get(array, index)     : Get an element from the array
<type>_set(array, index, value) : Set an element in the array
<type>_destroy(array)       : Destroy an array
```

The functions in this library are only low-level accessor functions designed to directly access C arrays. Like C, no bounds checking is performed so use at your own peril.

```
%}

// -----
// Integer array support
// -----

%subsection "Integer Arrays"
/* The following functions provide access to integer arrays (mapped
   onto the C 'int' datatype. */

%{
    ... Supporting C code ...
%}
int *int_array(int nitens);
/* Creates a new array of integers. nitens specifies the number of elements.
   The array is created using malloc() in C and new() in C++. */

void int_destroy(int *array);
/* Destroys the given array. */

int int_get(int *array, int index);
/* Returns the value of array[index]. */

int int_set(int *array, int index, int value);
/* Sets array[index] = value. Returns value. */

// -----
// Floating point
// -----

%subsection "Floating Point Arrays"
/* The following functions provide access to arrays of floats and doubles. */

%{
    .. Supporting C code ...
%}
double *double_array(int nitens);
/* Creates a new array of doubles. nitens specifies the number of elements.
   The array is created using malloc() in C and new() in C++. */

void double_destroy(double *array);
/* Destroys the given array. */

double double_get(double *array, int index);
/* Returns the value of array[index]. */

double double_set(double *array, int index, double value);
```

```

/* Sets array[index] = value. Returns value. */

float *float_array(int nitems);
/* Creates a new array of floats. nitems specifies the number of elements.
   The array is created using malloc() in C and new() in C++. */

void float_destroy(float *array);
/* Destroys the given array. */

float float_get(float *array, int index);
/* Returns the value of array[index]. */

float float_set(float *array, int index, float value);
/* Sets array[index] = value. Returns value. */

// -----
// Character strings
// -----

%subsection "String Arrays"

%text %{
The following functions provide support for the 'char **' datatype. This
is primarily used to handle argument lists and other similar structures that
need to be passed to a C/C++ function.
%}

#if defined(SWIGTCL)
%text %{
To convert from a Tcl list into a 'char **', the following code can be used :

    # $list is a list
    set args [string_array expr {[llength $list] + 1}]
    set i 0
    foreach a $list {
        string_set $args $i $a
        incr i 1
    }
    string_set $i ""
    # $args is now a char ** type
%}
#elif defined(SWIGPERL)

%text %{
To convert from a Perl list into a 'char **', code similar to the following
can be used :

    # @list is a list
    my $l = scalar(@list);
    my $args = string_array($l+1);
    my $i = 0;
    foreach $arg (@list) {
        string_set($args,$i,$arg);
        $i++;
    }
    string_set($args,$i,"");

(of course, there is always more than one way to do it)
%}

```

```

#elif defined(SWIGPYTHON)

%text %{
To convert from a Python list to a 'char **', code similar to the following
can be used :

    # 'list' is a list
    args = string_array(len(list)+1)
    for i in range(0,len(list)):
        string_set(args,i,list[i])
    string_set(args,len(list),"")
%}

#endif

%{
    ... Supporting C code ...
%}
char **string_array(int nitems);
/* Creates a new array of strings. nitems specifies the number of elements.
   The array is created using malloc() in C and new() in C++. Each element
   of the array is set to NULL upon initialization. */

void string_destroy(char *array);
/* Destroys the given array. Each element of the array is assumed to be
   a NULL-terminated string allocated with malloc() or new(). All of
   these strings will be destroyed as well. (It is probably only safe to
   use this function on an array created by string_array) */

char *string_get(char **array, int index);
/* Returns the value of array[index]. Returns a string of zero length
   if the corresponding element is NULL. */

char *string_set(char **array, int index, char *value);
/* Sets array[index] = value. value is assumed to be a NULL-terminated
   string. A string of zero length is mapped into a NULL value. When
   setting the value, the value will be copied into a new string allocated
   with malloc() or new(). Any previous value in the array will be
   destroyed. */

```

In this file, all of the declarations are placed into a new section. We specify formatting information for our section. Since this is a general purpose library file, we have no idea what formatting our parent might be using so an explicit declaration makes sure we get it right. Each comment contains preformatted text describing each function. Finally, in the case of the string functions, we are using a combination of conditional compilation and documentation system directives to produce language-specific documentation. In this case, the documentation contains a usage example in the target scripting language.

When processed through the ASCII module, this file will produce documentation similar to the following :

```

7. SWIG C Array Module
=====

#include array.i

```

This module provides scripting language access to various kinds of C/C++ arrays. For each datatype, a collection of four functions are created :

```
<type>_array(size)           : Create a new array of given size
<type>_get(array, index)     : Get an element from the array
<type>_set(array, index, value) : Set an element in the array
<type>_destroy(array)        : Destroy an array
```

The functions in this library are only low-level accessor functions designed to directly access C arrays. Like C, no bounds checking is performed so use at your own peril.

7.1. Integer Arrays

The following functions provide access to integer arrays (mapped onto the C 'int' datatype).

```
int_array(nitems)
    [ returns int * ]
    Creates a new array of integers. nitems specifies the number of elements.
    The array is created using malloc() in C and new() in C++.
```

```
int_destroy(array)
    [ returns void ]
    Destroys the given array.
```

```
int_get(array,index)
    [ returns int ]
    Returns the value of array[index].
```

```
int_set(array,index,value)
    [ returns int ]
    Sets array[index] = value. Returns value.
```

7.2. Floating Point Arrays

The following functions provide access to arrays of floats and doubles.

```
double_array(nitems)
    [ returns double * ]
    Creates a new array of doubles. nitems specifies the number of elements.
    The array is created using malloc() in C and new() in C++.
```

```
double_destroy(array)
    [ returns void ]
    Destroys the given array.
```

```
double_get(array,index)
    [ returns double ]
    Returns the value of array[index].
```

```
double_set(array,index,value)
    [ returns double ]
    Sets array[index] = value. Returns value.
```

```
float_array(nitems)
    [ returns float * ]
```

Creates a new array of floats. `nitems` specifies the number of elements. The array is created using `malloc()` in C and `new()` in C++.

```
float_destroy(array)
  [ returns void ]
  Destroys the given array.
```

```
float_get(array,index)
  [ returns float ]
  Returns the value of array[index].
```

```
float_set(array,index,value)
  [ returns float ]
  Sets array[index] = value. Returns value.
```

7.3. String Arrays

The following functions provide support for the 'char **' datatype. This is primarily used to handle argument lists and other similar structures that need to be passed to a C/C++ function.

To convert from a Python list to a 'char **', code similar to the following can be used :

```
# 'list' is a list
args = string_array(len(list)+1)
for i in range(0,len(list)):
    string_set(args,i,list[i])
string_set(args,len(list),"")
```

```
string_array(nitems)
  [ returns char ** ]
  Creates a new array of strings. nitems specifies the number of elements. The array is created using malloc() in C and new() in C++. Each element of the array is set to NULL upon initialization.
```

```
string_destroy(array)
  [ returns void ]
  Destroys the given array. Each element of the array is assumed to be a NULL-terminated string allocated with malloc() or new(). All of these strings will be destroyed as well. (It is probably only safe to use this function on an array created by string_array)
```

```
string_get(array,index)
  [ returns char * ]
  Returns the value of array[index]. Returns a string of zero length if the corresponding element is NULL.
```

```
string_set(array,index,value)
  [ returns char * ]
  Sets array[index] = value. value is assumed to be a NULL-terminated string. A string of zero length is mapped into a NULL value. When setting the value, the value will be copied into a new string allocated with malloc() or new(). Any previous value in the array will be destroyed.
```

ASCII Documentation

The ASCII module produces documentation in plaintext as shown in the previous example. Two formatting options are available (default values shown) :

```
ascii_indent = 8
ascii_columns = 70
```

'ascii_indent' specifies the number of characters to indent each function description. 'ascii_columns' specifies the width of the output when reformatting text.

When reformatting text, all extraneous white-space is stripped and text is filled to fit in the specified number of columns. The output text will be left-justified. A single newline is ignored, but multiple newlines can be used to start a new paragraph. The character sequence '\\\ ' can be used to force a newline.

Preformatted text is printed into the resulting output unmodified although it may be indented when used as part of a function description.

HTML Documentation

The HTML module produces documentation in HTML format (who would have guessed?). However, a number of style parameters are available (shown with default values)

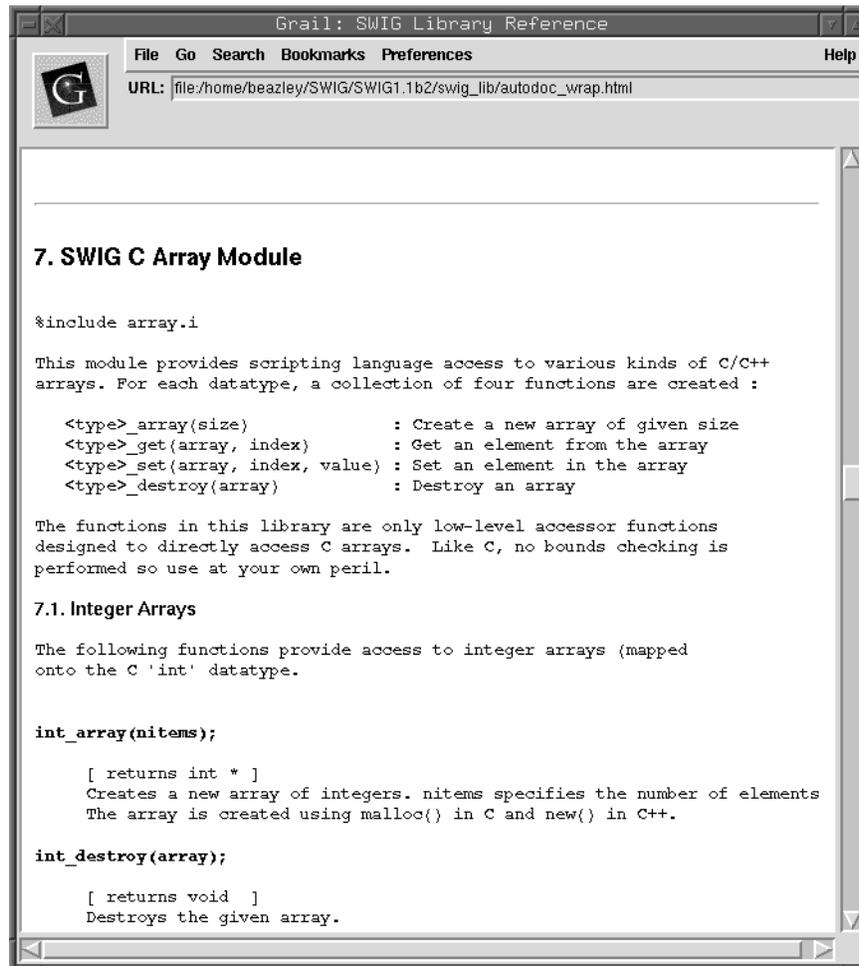
```
html_title = "<H1>:</H1>"
html_contents = "<H1>:</H1>"
html_section = "<HR><H2>:</H2>"
html_subsection = "<H3>:</H3>"
html_subsubsection = "<H4>:</H4>"
html_usage = "<B><TT>:</TT></B>"
html_descrip = "<BLOCKQUOTE>:</BLOCKQUOTE>"
html_text = "<P>"
html_cinfo = ""
html_preformat = "<PRE>:</PRE>"
html_body = "<BODY bg_color=\"#ffffff\">:</BODY>"
```

Any of these parameters can be changed, by simply specifying them after a %title or %section directive. However, the effects are applied globally so it probably makes sense to use the %style directive instead. For example :

```
%style html_contents="<HR><H1>:</H1>"
... Rest of declarations ...
```

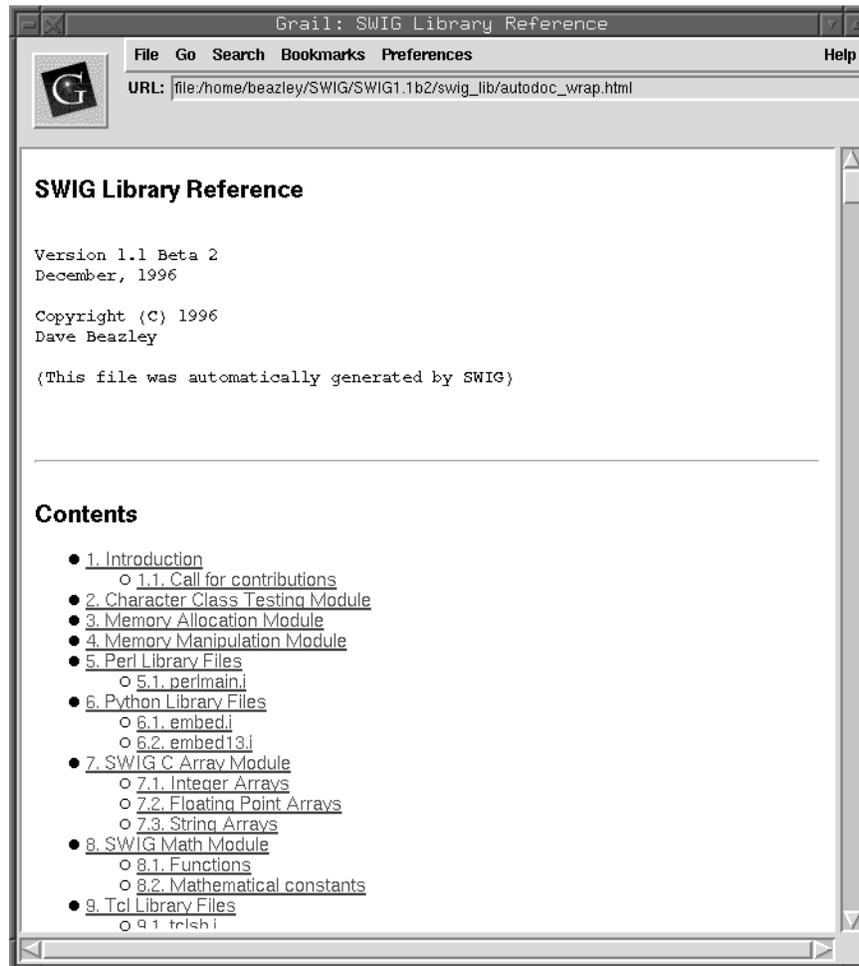
Each tag uses a ":" to separate the start and end tags. Any text will be inserted in place of the ":". Since strings are specified in SWIG using quotes, any quotes that need to be inserted into a tag should be escaped using the "\" character.

Sample HTML output is shown below :



Since our example used preformatted text, the output is very similar to the ASCII module. However, if you use the default mode, it is possible to insert HTML markup directly into your C comments for a more personalized document.

For navigation within the document, SWIG also produces a table of contents with links to each section within the document. With a large interface, the contents may look something like this :



LaTeX Documentation

The LaTeX module operates in a manner similar to the HTML module. The following style parameters are available (some knowledge of LaTeX is assumed).

```

latex_parindent = "0.0in"
latex_textwidth = "6.5in"
latex_documentstyle = "[11pt]{article}"
latex_oddsidemargin = "0.0in"
latex_pagestyle = "\\pagestyle{headings}"
latex_title = "{\\Large \\bf :} \\\\n"
latex_preformat = "{\\small \\begin{verbatim}:\\end{verbatim}}"
latex_usage = "{\\tt \\bf :}"
latex_descrip = "{\\\\n \\makebox[0.5in]{} \\begin{minipage}[t]{6in} : \\n
\\end{minipage} \\\\n";
latex_text = ":\\\\n"

```

```
latex_cinfo = "{\\tt : }"  
latex_section = "\\section{:}"  
latex_subsection = "\\subsection{:}"  
latex_subsubsection = "\\subsubsection{:}"
```

The style parameters, well, look downright ugly. Keep in mind that the strings used by SWIG have escape codes in them so it's necessary to represent the `'\'` character as `'\\'`. Thus, within SWIG your code will look something like this :

```
%style latex_section="\\newpage \n \\section{:}"
```

The default values should be sufficient for creating a readable LaTeX document in any case you don't want to worry changing the default style parameters.

C++ Support

C++ classes are encapsulated in a new subsection of the current section. This subsection contains descriptions of all of the member functions and variables. Since language modules are responsible for creating the documentation, the use of shadow classes will result in documentation describing the resulting shadow classes, not the lower level interface to the code.

While it's not entirely clear that this is the best way to document C++ code, it is a start (and it's better than no documentation).

The Final Word?

Early versions of SWIG used a fairly primitive documentation system, that could best be described as "barely usable." The system described here represents an almost total rewrite of the documentation system. While it is, by no means, a perfect solution, I think it is a step in the right direction. The SWIG library is now entirely self-documenting and is a good source of documentation examples. As always suggestions and improvements are welcome.