SWIG Reference Manual

Version 1.1
June, 1997

David M. Beazley
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
beazley@cs.utah.edu

Copyright (C) 1996,1997
All Rights Reserved

SWIG Reference Manual

Copyright (C) 1996, 1997
David M. Beazley
All Rights Reserved

You may distribute this document in whole provided this copyright notice is retained. Unauthorized duplication of this document in whole or part is prohibited without the express written consent of the author.

SWIG 1.1 is Copyright (C) 1995-1997 by the University of Utah and the Regents of the Universiy of California and is released under the following license :

This software is copyrighted by the University of Utah and the Regents of the University of California. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that (1) existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions and (2) redistributions including binaries reproduce these notices in the supporting documentation. No written agreement, license, or royalty fee is required for any of the authorized uses. Substantial modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHOR, THE UNIVERSITY OF CALIFORNIA, THE UNIVERSITY OF UTAH, OR THE DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS OR ANY OF THE ABOVE PARTIES HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS, THE UNIVERSITY OF CALIFORNIA, THE UNIVERSITY OF UTAH, AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Command Line Options

General Options

-c	Do not include the SWIG runtime functions. This includes the pointer type-checker and support code. Used when working with multiple SWIG generated module.
-c++	Enable C++ processing. Required for SWIG to understand C++ keywords and apply C++ specific processing.
-ci	Check a file into the SWIG library. The file is placed into the directory as used by the target scripting language. Must have write permission on the SWIG library.
-co	Check a file out of the SWIG library and place it in the current directory. Does nothing if the file already exists.
-debug	SWIG debugging module. Doesn't do much of anything interesting.
-D*symbol*	Defines a symbol (for conditional compilation). Roughly equivalent to the C compiler version.
-d *docfile*	Write documentation to docfile. The filename will have the appropriate suffix appended to it (.html, .doc, .tex, etc...).
-dascii	Produce ASCII documentation.
-dhtml	Produce HTML documentation.
-dlatex	Produce LaTeX documentation.
-dnone	Produce no documentation.
-guile	Make wrappers for FSF Guile 1.0.
-help	Display all available command line options.
-I*dir*	Specify a new directory to search for SWIG library files or any files included with the SWIG %include directive.
-l*file*	Specifies additional files to wrap as part of this module. Files specified with `-l` are simply appended to the input file and wrapped as part of the same module.
-make_default	Make default constructors and destructors for all classes and structures without them. Works for both C and C++.
-module *name*	Change the module name. Overrides `%module`.

-nocomment	Ignore all comments (when producing documentation)
-o *file*	Change the name of the output file.
-objc	Enable Objective-C processing
-perl4	Perl4 module.
-perl5	Perl5 module.
-python	Python module.
-stat	Produce statistics. Somewhat inaccurate at the moment, but this option also produces other diagnostic information.
-strict *n*	Change strictness of pointer type-checking. 0 = no checking, 1 = warnings only, 2 = strict (the default)
-swiglib	Prints location of SWIG library and exits.
-t *typemap_file*	Use a typemap file. At most only one typemap can be specified. This file is read before any other files.
-tcl	Tcl module. Compatible with Tcl 7.x and Tcl 8.x.
-tcl8	Tcl 8.0 module. Uses the new Tcl 8.0 object interface.
-v	(Very) Verbose mode. Prints out everything SWIG is doing, where it's looking for files, etc... Useful for tracking down SWIG installation problems or compilation problems.
-version	Display SWIG's version number.

Comment handling

-Safter	Assume comments appear after each declaration.
-Sbefore	Assume comments appear before each declaration.
-Schop_bottom *n*	Number of lines to chop off the bottom of comment blocks.
-Schop_left *n*	Number of characters to chop off the left of comments
-Schop_right *n*	Number of characters to chop off the right of comments.
-Schop_top *n*	Number of lines to chop off top of comment blocks.
-Signore	Ignore comments.
-Sskip *n*	Set maximum number of lines between comments and declarations (default = 1).

-Stabify	Leave tabs intact.
-Suntabify	Expand comment tabs into spaces (default).

Documentation Processing

-Sformat	Allow SWIG to reformat documentation text (default).
-Sinfo	Print C calling information for each declaration (default)
-Snoinfo	Omit C calling information.
-Snosort	Do not sort the documentation (default).
-Spre	Assume text is preformatted.
-Ssort	Sort the documentation.

Tcl Options (available with -tcl or -tcl8)

-prefix ***name***	Set a prefix to be appended to all names.
-htcl ***tcl.h***	Specify the name of the "tcl.h" header file. Using this is likely to produce modules that are incompatible with other versions of Tcl however.
-htk ***tk.h***	Specify the name of the "tk.h" header file.
-namespace	Build module into a [incr Tcl] namespace. The name of the namespace is the same as specified with %module or can be set using the -prefix option.
-plugin	Produce code compatible with Tcl Netscape Plugin
-noobject	Omit code for object oriented interface. Results in smaller files.
-old	Use old SWIG interface (same as -noobject).

Perl5 Options (available with -perl5)

-exportall	Export all symbols (not generally recommended)
-package ***name***	Set package name (if different than the module).
-static	Omit code related to dynamic loading.
-shadow	Create shadow classes.

Python Options (available with -python)

-globals ***name***	Set name used to access C global variables ('cvar' by default).

`-shadow`	Generate shadow classes.
`-docstring`	Create doc strings (only works with -shadow).

Perl4 Options (available with -perl4)

`-prefix` ***`name`***	Set a prefix to be appended to all names

SWIG Directives

%{ ... %}

Code insertion block. Everything enclosed in %{, %} is copied verbatim into the resulting wrapper code. Primarily used to include header files and helper functions. For example :

```
%{
#include <GL/gl.h>
#include <GL/glx.h>
%}
```

Most SWIG input files contain at least one of these declarations. Any number of code insertion blocks may appear in an interface file.

%addmethods [*classname*] { ... }

Adds additional "methods" to C++ classes, C structures, and Objective-C interfaces. This can be used to provide "additional" functionality to objects when building a scripting language interface. It can also be used to make C programs appear object oriented. The directive can be used in a number of ways :

```
struct Vector {
       double x,y,z;
       %addmethods {
              Vector() { return (Vector *) malloc(sizeof(Vector)); }
              ~Vector() { free(self); }
              void output() {
                     printf("[ %g, %g, %g ]\n", self->x, self->y, self->z);
              }
       }
};
```

When this is wrapped by SWIG, Vector will not only be a C struct, but will have member functions added to it. For example (in Python) :

```
>>> v = Vector()
>>> v.x = 3
>>> v.y = 20
>>> v.z = -5
>>> v.output()
[ 3, 20, -5 ]
>>> del v
```

%addmethods may also be detached from the original class definition as in :

```
class List {         // Original C++ class
public:
       ...
};

// Now add methods to the class
%addmethods List {
       void output(FILE *f);
```

```
        int size();
};
```

If the code for the added methods is not supplied, the methods are expanded into function calls of the format `Classname_method(Class *obj, ...)`. For the List class above, you would need to implement the following functions :

```
void List_output(List *l, FILE *f) {
        // Added method List::output
}

int List_size(List *l) {
        // Added method List::size
        return result;
}
```

Added methods are not part of the real C++ definition--nor does SWIG modify the original C++ class or C structure in any way. However, added methods can be inherited and used in derived classes (when used within SWIG).

%apply *datatype* { *typelist* };

Changes the processing of a datatype by applying a typemap rule to it. This can be used to make arguments act as output values, input values, etc... For example :

```
%include typemaps.i

%apply double *INPUT { double *in1, double *in2};
%apply double *OUPUT { double *out, double *result, Real *ans };

void add(double *in1, double *in2, double *result);

%include constraints.i
%apply Number POSITIVE { double x };

double log(double x);
```

This directive is primarily used in conjunction with SWIG library files (such as `typemaps.i` or `constraints.i`). The effects of the `%apply` directive remain in effect until subsequent calls to `%apply` or the `%clear` directive is issued.

%checkout *filename*

Extracts a file from the SWIG library and puts it in the current directory. Will not overwrite an existing file if it exists. This function is primarily designed to support the SWIG library. If a particular library file requires the use of a particular Tcl/Python/Perl script, the script can be placed in the library and checked out into the user's working directory whenever the library file is used. For example :

```
%checkout array.tcl          // Copy 'array.tcl' into current directory
%checkout "database.pl"      // Copy 'database.pl' into current directory
```

%clear *datatype [, typelist]*;

The opposite of %apply. Clears any special processing that has been applied to a datatype. For example :

```
%clear double *in;
%clear double *output, double *result, Real *ans;
```

This only undos the effect of the %apply directive. It does not change any special processing that might have been added explicitly using the %typemap directive.

%disabledoc

Disables all documentation generation until explicitly re-enabled using the %enabledoc directive. Can be safely nested (i.e. use of %disabledoc/%enabledoc when documentation is already disabled has no effect).

%echo "*message*"

Prints a message to the console when compiled. Modules can use this to print diagnostic or informative message to the screen when SWIG is run. For example :

```
%echo "Expect a warning here."
double foo(double a[4]);
```

One can also use the following form if printing alot of text

```
%echo %{
... a block of text ...
%}
```

%elif *expr*

Conditional compilation. Is the same as #elif expr. Currently the implementation only allows a single type of expression as shown :

```
#elif defined(FOO)
...

#elif !defined(FOO)
...
```

For practical purposes, the #elif version should be used in most interfaces. The %elif form is sometimes used by preprocessing tools (since %elif passes through the C preprocessor without modification).

%else

Conditional compilation. Is the same as #else. The %else form is only used by some special preprocessing tools (since it passes through the C preprocessor unchanged).

%enabledoc

Enables the documentation system after being disabled by %disabledoc.

%endif

Conditional compilation. Same as #endif.

%except(*lang*) { ... }

Creates a user-definable exception handler. For example :

```
%except(perl5) {
       try {
              $function
       } catch (RangeError) {
              croak("RangeError");
       } catch (OutofMemory) {
              croak("Out of memory");
       } catch(...) {
              croak("Unknown exception thrown");
       }
}
```

The language specifier indicates which language the handler is for. If it doesn't match the target language, the exception handler is simply ignored. The code specified gets included into all SWIG generated wrapper functions. The $function variable gets replaced by the real C/C++ function call when code is generated.

Exception handlers can be redefined as needed by simply specifying a new exception handler. To clear an exception, use %except with no code. For example :

```
%except(perl5);       // Clears any exception handlers
```

A shorthand version of %except is available for language-independent processing as follows :

```
%include exception.i

%except {
       try {
              $function
       } catch (RangeError) {
              SWIG_exception(SWIG_ValueError, "Range Error");
       } catch (OutofMemory) {
              SWIG_exception(SWIG_MemoryError,"Out of memory");
       } catch(...) {
              SWIG_exception(SWIG_RuntimeError,"Unknown exception");
       }
}
```

When used in this manner, the exception handler will be applied to all target languages.

%extern *filename*

Includes a file, but only extracts type-definitions. This includes typedefs, class, and structure definitions. None of the C declarations in the file are converted into wrapper code. Use this if your interface relies on a common set of definitions. Examples :

```
%extern common.i
%extern "types.h"
```

%if *expr*

Conditional compilation. Is identical to the `#if` directive. Does not currently allow arbitrary expressions, but does support the C `defined()` macro. Use `#if` for most applications. `%if` is usually only used by special preprocessing tools. Examples :

```
#if defined(FOO)
#if !defined(FOO)
```

%ifdef *symbol*

Conditional compilation. Is the same as `#ifdef`. Checks to see if a particular symbol has been defined within the SWIG parser. The `%ifdef` form is usually only used by special preprocessing tools, but is functionally identical.

```
#ifdef SWIG
...
#endif

%ifdef SWIG
...
%endif
```

%ifndef *symbol*

Conditional compilation. Is the same as `#ifndef`.

%import *filename*

Imports all of the type information and definitions from another SWIG module. Is functionally the same as `%extern`, but tells SWIG that all of the definitions are located in a separately compiled module. Does not generate wrappers for any of the declarations in the imported file. Examples :

```
%import graphics.i
%import "network.i"
```

%include *filename*

Copies a separate file into the current interface file and parses it. All definitions and declarations are processed exactly as if they had been inlined into the current file. `%include` is primarily used for building interfaces to various components, including library files,

and creating packages. For example :

```
%module opengl

%include gl.i
%include glu.i
%include "glx.i"
%include "glaux.i"
```

Include can also be used to process header and C source files :

```
%include "gd.h"
%include "methods.c"
```

To locate files, `%include` manages a search-path of directories. Additional directories can be added by running SWIG with the `-Idir` option.

%init %{ ... %}

All SWIG generated modules contain an initialization function to bind all of the wrapper functions to the target scripting language. The `%init` directive inserts code into this initialization function. For example :

```
%init %{
        printf("Initializing my module...\n");
        module_initialize();
%}
```

This would call a C function `module_initialize()` whenever the resulting module was loaded. `%init` is typically used to perform custom initializations and special processing at module startup.

%inline %{ ... %}

Includes and wraps all of the declarations inside the `%{, %}` block. Primarily used to create special helper functions in your scripting language. For example,

```
%inline %{
        double *double_array(int size) {
                return (double *) malloc(size*sizeof(double));
        }
%}
```

Creates both a new function called `double_array()` and a scripting language wrapper. The `%inline` directive is really short-hand for the following equivalent code :

```
%{
double *double_array(int size) {
        return (double *) malloc(size*sizeof(double));
}
%}

double *double_array(int size);
```

It is illegal for any SWIG directives to appear inside the code given to %inline.

%localstyle *stylelist*

Changes the documentation style for the current section. stylelist is a comma separated list of style parameters. For example :

```
%localstyle sort, pre, skip=2
```

The directive is often unnecessary because the style parameters can be specified at the start of each section. See the %section, %subsection, %subsubsection directives for details. A full list of style parameters is described in the "Documentation system" section of this manual.

%module *name [, modlist]*

Sets the name of the module. This directive should appear at the beginning of an interface file and before any C declarations. Subsequent occurences of %module will be ignored. The directive will also be ignored if it appears after any C declarations (the module name will default to "swig" in this case). The modlist parameter is used to specify the initialization of other modules and is only used when building statically linked executables. Examples :

```
%module example
... C declarations ...

// Create a module 'package' and initialize the other modules listed.
%module package, graphics, network, visual, fileio
... C declarations ...
```

%native(*name*) *function*

Add a natively written wrapper function to the module. For example, if you've already written extension functions for Tcl, Python, Perl, etc... they can be added as follows :

```
%native(foo) wrap_foo;
%native(bar) int wrap_bar(Tcl_Interp *, ClientData, int argc, char *argv[]);
%native(spam) PyObject *pyspam(PyObject *self, PyObject *other);
```

SWIG doesn't actually look at the function parameters when adding a native method. The only processing performed is the creation of a "command" to call your native method. The name of the command is always specified in paranthesis when using the %native directive.

%name(*newname*) *decl*

Tells SWIG to rename a declaration when creating a scripting language command. For example ,

```
%name(output) void print(Matrix *m);
```

binds the `print()` function to a scripting language command "output". `%name` can also be applied to C++ class members and classes. For example :

```
class Foo {
public:
       Foo();
       ~Foo();
       // Rename a member function
       %name(spam) double foo(int a);
};

// Renames class "Foo" to class "Bar"
%name(Bar) class Foo {
public:
       ...
};
```

%new *decl*

Gives a hint to the language modules that a function is returning newly allocated memory.. For example :

```
/* A "leaky" C function */
char *spam() {
       char *c = (char *) malloc(200);
       sprintf(c,"Hello world");
       return c;
}

// SWIG interface file
%new char *spam();
```

When implemented properly, language modules will return the result to the scripting language, and properly cleanup or manage the returned memory. However, it should be cautioned that not all language modules handle `%new` in the same manner or for all datatypes. You can provide your own handler by specifying a typemap however.

%pragma(*lang*) *var* [= *value*];

Passes a language specific hint directly to the SWIG language modules. `lang` specifies the language. For example :

```
%pragma(python) code="from spam import *";
%pragma(perl5) include="support.pl";
%pragma(tcl) volatile;
```

The precise meaning of pragma directives is highly language specific.

%readonly

Enables read-only mode. All variables and class members will be read-only until the directive is explicitly disabled using `%readwrite`. Examples :

```
// Create some read-only variables
```

```
%readonly
int Status;
char *error_msg;
%readwrite

// Create some read-only class member
class List {
public:
       %readonly
       int length;
       %readwrite
       List();
       ~List();
       void insert(char *);
       void remove(char *);
       char *get(int i);
};
```

%readwrite

Disables read-only mode.

%rename *oldname newname;*

Applies a global renaming operation in which all future occurences of oldname will be replaced by newname. This renaming applies to the names of C functions, variables, classes, structures, member functions, and member data. `%rename` is a stronger, but functionality equivalent version of the `%name()` directive (it does the same thing but is applied to multiple occurrences of the same name). To disable %rename, simply give a second declaration of %rename to change the name back to the original version.

%section "*name*" [, *stylelist*]

Documentation system. Starts a new section with given name. An optional list of style parameters can be given at the end. For example :

```
%section "Graphics"
%section "Network",sort,pre
```

Style parameters only apply to this particular section and will be active until the next section is specified.

%subsection "*name*" [, *stylelist*]

Documentation system. Creates a new subsection with an optional list of style parameters. For example :

```
%subsection "2D Graphics"
%subsection "Sockets", sort, skip=1
```

It is illegal to create a `%subsection` without first creating a section. Subsections inherit all of the style parameters from their parent section.

%subsubsection "*name*" [, *stylelist*]

Documentation system. Creates a new subsubsection. Same rules as for %subsection.

%style *stylelist*

Documentation system. Set documentation styles globally for an entire interface file. For example :

```
%style sort, format
%style pre, html_title="<H1>:</H1>", html_section="<HR><H2>:</H2>", noinfo
```

Multiple occurrences of this directive may cause unpredictable behavior.

%text %{ ... %}

Documentation system. Creates a block of descriptive text for inclusion in documentation. For example :

```
%section "Graphics"

%text %{
The following functions can be used for 2D and 3D graphics.
%}

// Now put declarations
```

The %text directive can be used to add documentation that is not associated with any particular declaration.

%title "*name*" [, *stylelist*]

Documentation system. Sets the title of the documentation. When used, this should be the first directive in an interface file. For example :

```
%title "OpenGL module"
%module opengl

// Declarations ...
```

Repeated occurrences of the %title directive are ignored. If no title is specified, SWIG picks one based upon the module name.

%typedef *datatype name*;

A special version of the C typedef directive. SWIG understands both the normal typedef and %typedef. However, %typedef results in a new typedef being created (by SWIG) in the wrapper code. This is primarily used to work around some type-handling problems and renaming of datatypes in interface files. For example :

```
typedef double Real;         // Normal typedef. SWIG does nothing.
%typedef double Real;        // SWIG creates a 'typedef double Real' statement
                             // in the wrapper code.
%typedef int (*IFUNC)(int a, int b);
```

The `%typedef` directive is really just shorthand for the following SWIG code :

```
%{
typedef double Real;         // Include a typedef in the C code
%}

typedef double Real;         // Tell SWIG about it
```

Or

```
%inline %{
typedef double Real;
%}
```

%typemap(*lang*,*method*) *datatype* [, *datatype list*] { ... }

Creates a new typemap. Typemaps are used to modify SWIG's code generator by specifying special processing rules to certain C datatypes. Here are a few simple examples :

```
%typemap(tcl,in) int {
       $target = atoi($source);
}

%typemap(tcl,in) double input {
       $target = atof($source);
}
```

The `lang` parameter specifies the target language (tcl, perl5, python, perl4, guile). The `method` parameter determines the particular conversion (all of which have names). In this case "in" refers to the handling of input parameters. The datatype specifies which types the typemap will be applied to. The typemap will also be applied to any additional datatypes are given as a comma separated list. The matching process is name-based so in the above example, the first typemap will be applied to all integers, while the second typemap will only be applied to function parameters matching "`double input`".

The conversion code is inlined into the resulting wrapper code. The `$source` and `$target` variables are replaced with the real C variables containing the input and output of the type conversion.

A typemap may be applied to a list of datatype as follows :

```
%typemap(tcl,in) int, short, long {
       $target = ($type) atol($source);
}
```

In this case, the `$type` variable gets filled in with the real datatype being used.

Typemaps may declare persistent variables as follows :

```
%typemap(tcl,in) double *input(double temp) {
       temp = atof($source);
       $target = &temp;
}
```

Local temporaries may also be declared inside the code block, but these variables only exist inside the typemap code (and are destroyed upon completion).

A typemap may be deleted by specifying it with no conversion code. For example :

```
// Delete some typemaps
%typemap(tcl,in) double;
%typemap(tcl,in) int, short, long;
```

Typemaps are primarily designed for advanced users who want to customize SWIG. Please see the typemaps section of this manual for more details.

%wrapper %{ ... %}

Inserts code into the wrapper portion of the output file. This is almost never required, but is used by a few library files. Examples :

```
%wrapper %{
       /* Your C code here */
%}
```

Documentation style options

The following options are available when specifying documentation styles with the `%title`, `%section`, `%subsection`, `%subsubsection`, `%localstyle`, and `%style` directives.

Comment processing

`after`	Assume comments appear after a declaration.
`before`	Assume comments appear before a declaration
`chop_top=nlines`	Number of lines to strip from top of a comment block
`chop_bottom=nlines`	Number of lines to strip from bottom of a comment block
`chop_left=nlines`	Number of characters to strip from left of comment block
`chop_right=nlines`	Number of characters to strip from right of comment block
`format`	Allow SWIG to reformat text
`ignore`	Ignore comments
`info`	Print C information text
`keep`	Keep comments
`noinfo`	Don't print C information text
`nosort`	Don't sort documentation.
`pre`	Assume comments are preformatted.
`skip=nlines`	Number of blank lines between declarations and comments.
`sort`	Sort documentation.
`tabify`	Leave tabs intact.
`untabify`	Convert tabs into spaces.

ASCII Documentation

`ascii_indent=8`	Default indentation of function descriptions.
`ascii_columns=70`	Maximum width of output when reformatting.

HTML Documentation

```
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>"
```

LaTeX Documentation

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

Typemap Methods

The following typemap methods are available for modifying SWIG's output. The '$' variables listed for each typemap represent variables that are supplied by SWIG during code generation.

%typemap(lang,arginit)

Is used to assign function arguments to some default value. This may be useful in writing other type-handling functions that need to know if arguments were initialized or parsed correctly.

```
$source             Function argument (C representation)
$type               C datatype
$mangle             String representation of datatype
$basetype           Base C datatype
```

%typemap(lang,argout)

Returns values through function arguments.

```
$source             Function argument (C representation)
$target             The result object that will be returned.
$type               C datatype
$mangle             String representation of datatype
$basetype           Base C datatype
$arg                Original function argument (Scripting representation)
$argnum             Argument number
$name               Function name
```

%typemap(lang,check)

Checks validity of function arguments in their C representation.

```
$source             Scripting language representation (typically unused)
$target             Function argument (C representation)
$type               C datatype
$mangle             String representation of datatype
$basetype           Base C datatype
$arg                Original function argument
$argnum             Argument number
$name               Function name
```

%typemap(lang, const)

Code to create constant values (Tcl module only). Code is inserted into the module initialization function.

```
$source             Value of the constant
$target             Name of the constant
$value              Value of constant
```

%typemap(lang, default)

Supplies a default argument to certain function parameters.

```
$target             C variable corresponding to the function argument
$type               C datatype
$basetype           Base C datatype
$name               Function name
```

%typemap(lang, except)

Defines an exception handler. The typemap is applied to the return type and name of a function.

```
$function           Actual C function call
$source             Return result of the C function.
$name               Name of the function
$type               Return type of the function
$basetype           Base C datatype of function.
```

%typemap(lang,freearg)

Cleans up the function arguments. This code is executed after a function call to release any resources that might have been used in allocating function arguments.

```
$source             Function argument (C representation)
$target             Original funtion argument (Scripting representation)
$type               C datatype
$mangle             String representation of datatype.
$basetype           Base C datatype
$arg                Original function argument (Scripting representation)
$argnum             Argument number
$name               Function name
```

%typemap(lang, ignore)

Tells SWIG how to ignore an argument. This effectively creates a “hidden” argument. Since all arguments are needed to make the real C function call, this typemap is used to assign a default value to the argument.

```
$target             C variable corresponding to the function argument
$type               C datatype
$basetype           Base C datatype
$name               Function name
```

%typemap(lang,in)

Converts function arguments from a scripting language representation to a C representation.

```
$source             Scripting language object
$target             C variable to hold result
$type               C datatype
$mangle             String representation of C datatype
```

```
$basetype           Base C datatype (without any pointers).
$name               Function name
$argnum             Argument number
```

%typemap(lang,memberin)

Sets the value of structure and class members. Is implemented in a language independent manner so this typemap should not contain any scripting language specific code.

```
$source             C variable containing the input value
$target             Structure or class member
$type               C datatype
$basetype           Base C datatype
$name               Member name
```

%typemap(lang, memberout)

Retrieves the value of a structure or class member. Is implemented in a language independent manner so this typemap should not contain any scripting language specific code.

```
$source             Structure or class member
$target             C datatype to return result in
$type               C datatype
$basetype           Base C datatype
$name               Member name
```

%typemap(lang, newfree)

Provides code to handle function return values that have been specified with the SWIG %new directive.

```
$source             C variable containing the result
$type               C datatype
$basetype           Base C datatype
$name               Name of the C function
```

%typemap(lang,out)

Converts the result of a C function to a scripting language object.

```
$source             Result of C function
$target             Scripting language object
$type               C datatype
$mangle             String representation of datatype.
$basetype           Base C datatype
$name               Function name
```

%typemap(lang,ret)

Cleans up the return result of a function. This code is executed immediately before returning control back to the scripting language.

```
$source             Result of the C function
```

```
$type               C datatype
$mangle             String representation of datatype
$basetype           Base C datatype
$name               Function name
```

%typemap(lang,varin)

Sets the value of a C global variable. Not supported in all languages.

```
$source             Scripting language object
$target             C global variable
$type               C datatype
$mangle             String representation of datatype
$basetype           Base C datatype
$name               Name of the variable.
```

%typemap(lang,varout)

Retrieves the value of a C global variable. Not supported by all languages.

```
$source             C Global variable
$target             Scripting language object
$type               C datatype
$mangle             String representation of datatype
$basetype           Base C datatype
$name               Name of the C variable.
```