



## **SWIG–1.3 Documentation**

# Table of Contents

<b><u>SWIG-1.3 Development Documentation</u></b>	<b>1</b>
<u>Sections</u>	1
<u>SWIG Core Documentation</u>	1
<u>Language Module Documentation</u>	1
<u>Developer Documentation</u>	1
<u>Documentation that has not yet been updated</u>	1
<b><u>1 Preface</u></b>	<b>2</b>
1.1 <u>Introduction</u>	2
1.2 <u>Special Introduction for Version 1.3</u>	2
1.3 <u>SWIG Versions</u>	2
1.4 <u>SWIG resources</u>	2
1.5 <u>Prerequisites</u>	3
1.6 <u>Organization of this manual</u>	3
1.7 <u>How to avoid reading the manual</u>	3
1.8 <u>Backwards Compatibility</u>	3
1.9 <u>Credits</u>	4
1.10 <u>Bug reports</u>	4
<b><u>2 Introduction</u></b>	<b>5</b>
2.1 <u>What is SWIG?</u>	5
2.2 <u>Why use SWIG?</u>	5
2.3 <u>A SWIG example</u>	6
2.3.1 <u>SWIG interface file</u>	6
2.3.2 <u>The swig command</u>	7
2.3.3 <u>Building a Perl5 module</u>	7
2.3.4 <u>Building a Python module</u>	7
2.3.5 <u>Shortcuts</u>	8
2.4 <u>Supported C/C++ language features</u>	8
2.5 <u>Non-intrusive interface building</u>	9
2.6 <u>Incorporating SWIG into a build system</u>	9
2.7 <u>Hands off code generation</u>	9
2.8 <u>SWIG and freedom</u>	9
<b><u>3 Getting started on Windows</u></b>	<b>11</b>
3.1 <u>Installation on Windows</u>	11
3.1.1 <u>Windows Executable</u>	11
3.2 <u>SWIG Windows Examples</u>	11
3.2.1 <u>Instructions for using the Examples with Visual Studio</u>	11
3.2.1.1 <u>Python</u>	12
3.2.1.2 <u>TCL</u>	12
3.2.1.3 <u>Perl</u>	12
3.2.1.4 <u>Java</u>	12
3.2.1.5 <u>Ruby</u>	12
3.2.1.6 <u>C#</u>	13
3.2.2 <u>Instructions for using the Examples with other compilers</u>	13
3.3 <u>SWIG on Cygwin and MinGW</u>	13
3.3.1 <u>Building swig.exe on Windows</u>	13
3.3.1.1 <u>Building swig.exe using MinGW and MSYS</u>	13
3.3.1.2 <u>Building swig.exe using Cygwin</u>	13
3.3.1.3 <u>Building swig.exe alternatives</u>	13
3.3.2 <u>Running the examples on Windows using Cygwin</u>	13

# Table of Contents

<b>4 Scripting Languages</b>	<b>14</b>
4.1 The two language view of the world	14
4.2 How does a scripting language talk to C?	14
4.2.1 Wrapper functions	15
4.2.2 Variable linking	15
4.2.3 Constants	16
4.2.4 Structures and classes	16
4.2.5 Proxy classes	17
4.3 Building scripting language extensions	17
4.3.1 Shared libraries and dynamic loading	17
4.3.2 Linking with shared libraries	18
4.3.3 Static linking	18
<b>5 SWIG Basics</b>	<b>19</b>
5.1 Running SWIG	20
5.1.1 Input format	20
5.1.2 SWIG Output	21
5.1.3 Comments	21
5.1.4 C Preprocessor	21
5.1.5 SWIG Directives	21
5.1.6 Parser Limitations	22
5.2 Wrapping Simple C Declarations	23
5.2.1 Basic Type Handling	23
5.2.2 Global Variables	24
5.2.3 Constants	25
5.2.4 A brief word about const	26
5.2.5 A cautionary tale of char *	27
5.3 Pointers and complex objects	27
5.3.1 Simple pointers	27
5.3.2 Run time pointer type checking	28
5.3.3 Derived types, structs, and classes	28
5.3.4 Undefined datatypes	29
5.3.5 Typedef	29
5.4 Other Practicalities	30
5.4.1 Passing structures by value	30
5.4.2 Return by value	30
5.4.3 Linking to structure variables	31
5.4.4 Linking to char *	31
5.4.5 Arrays	32
5.4.6 Creating read-only variables	33
5.4.7 Renaming and ignoring declarations	34
5.4.8 Default/optional arguments	35
5.4.9 Pointers to functions and callbacks	35
5.5 Structures and unions	37
5.5.1 Typedef and structures	38
5.5.2 Character strings and structures	38
5.5.3 Array members	39
5.5.4 Structure data members	39
5.5.5 C constructors and destructors	40
5.5.6 Adding member functions to C structures	40
5.5.7 Nested structures	43
5.5.8 Other things to note about structure wrapping	44
5.6 Code Insertion	45
5.6.1 The output of SWIG	45

# Table of Contents

## 5 SWIG Basics

5.6.2 Code insertion blocks.....	45
5.6.3 Inlined code blocks.....	46
5.6.4 Initialization blocks.....	46
5.7 An Interface Building Strategy.....	46
5.7.1 Preparing a C program for SWIG.....	46
5.7.2 The SWIG interface file.....	47
5.7.3 Why use separate interface files?.....	47
5.7.4 Getting the right header files.....	48
5.7.5 What to do with main().....	48

## 6 SWIG and C++.....49

6.1 Comments on C++ Wrapping.....	49
6.2 Approach.....	50
6.3 Supported C++ features.....	50
6.4 Command line options and compilation.....	51
6.5 Simple C++ wrapping.....	51
6.5.1 Constructors and destructors.....	51
6.5.2 Default constructors.....	52
6.5.3 When constructor wrappers aren't created.....	52
6.5.4 Copy constructors.....	53
6.5.5 Member functions.....	54
6.5.6 Static members.....	54
6.5.7 Member data.....	54
6.6 Default arguments.....	55
6.7 Protection.....	56
6.8 Enums and constants.....	56
6.9 Friends.....	57
6.10 References and pointers.....	57
6.11 Pass and return by value.....	58
6.12 Inheritance.....	59
6.13 A brief discussion of multiple inheritance, pointers, and type checking.....	61
6.14 Renaming.....	62
6.15 Wrapping Overloaded Functions and Methods.....	62
6.15.1 Dispatch function generation.....	63
6.15.2 Ambiguity in Overloading.....	64
6.15.3 Ambiguity resolution and renaming.....	65
6.15.4 Comments on overloading.....	69
6.16 Wrapping overloaded operators.....	69
6.17 Class extension.....	70
6.18 Templates.....	71
6.19 Namespaces.....	78
6.20 Exception specifiers.....	82
6.21 Pointers to Members.....	83
6.22 Smart pointers and operator->().....	83
6.23 Using declarations and inheritance.....	85
6.24 Partial class definitions.....	87
6.25 A brief rant about const-correctness.....	87
6.26 Proxy classes.....	88
6.26.1 Construction of proxy classes.....	88
6.26.2 Resource management in proxies.....	89
6.26.3 Language specific details.....	90
6.27 Where to go for more information.....	91

# Table of Contents

<b>7 Preprocessing</b>	<b>92</b>
<a href="#">7.1 File inclusion</a>	92
<a href="#">7.2 File imports</a>	92
<a href="#">7.3 Conditional Compilation</a>	92
<a href="#">7.4 Macro Expansion</a>	93
<a href="#">7.5 SWIG Macros</a>	94
<a href="#">7.6 C99 and GNU Extensions</a>	94
<a href="#">7.7 Preprocessing and <code>%{ ... }%</code> blocks</a>	95
<a href="#">7.8 Preprocessing and <code>{ ... }</code></a>	95
<a href="#">7.9 Viewing preprocessor output</a>	95
<b>8 SWIG library</b>	<b>96</b>
<a href="#">8.1 The <code>%include</code> directive and library search path</a>	96
<a href="#">8.2 C Arrays and Pointers</a>	96
<a href="#">8.2.1 <code>cpointer.i</code></a>	96
<a href="#">8.2.2 <code>carrays.i</code></a>	98
<a href="#">8.2.3 <code>cmalloc.i</code></a>	100
<a href="#">8.2.4 <code>cdata.i</code></a>	101
<a href="#">8.3 C String Handling</a>	102
<a href="#">8.3.1 Default string handling</a>	102
<a href="#">8.3.2 Passing binary data</a>	103
<a href="#">8.3.3 Using <code>%newobject</code> to release memory</a>	103
<a href="#">8.3.4 <code>cstring.i</code></a>	103
<a href="#">8.4 C++ Library</a>	107
<a href="#">8.4.1 <code>std_string.i</code></a>	107
<a href="#">8.4.2 <code>std_vector.i</code></a>	108
<a href="#">8.5 Utility Libraries</a>	110
<a href="#">8.5.1 <code>exception.i</code></a>	110
<b>9 Argument Handling</b>	<b>111</b>
<a href="#">9.1 The <code>typemaps.i</code> library</a>	111
<a href="#">9.1.1 Introduction</a>	111
<a href="#">9.1.2 Input parameters</a>	112
<a href="#">9.1.3 Output parameters</a>	113
<a href="#">9.1.4 Input/Output parameters</a>	114
<a href="#">9.1.5 Using different names</a>	114
<a href="#">9.2 Applying constraints to input values</a>	115
<a href="#">9.2.1 Simple constraint example</a>	115
<a href="#">9.2.2 Constraint methods</a>	115
<a href="#">9.2.3 Applying constraints to new datatypes</a>	115
<b>10 Typemaps</b>	<b>116</b>
<a href="#">10.1 Introduction</a>	117
<a href="#">10.1.1 Type conversion</a>	117
<a href="#">10.1.2 Typemaps</a>	118
<a href="#">10.1.3 Pattern matching</a>	119
<a href="#">10.1.4 Reusing typemaps</a>	120
<a href="#">10.1.5 What can be done with typemaps?</a>	120
<a href="#">10.1.6 What can't be done with typemaps?</a>	121
<a href="#">10.1.7 The rest of this chapter</a>	122
<a href="#">10.2 Typemap specifications</a>	122
<a href="#">10.2.1 Defining a typemap</a>	122
<a href="#">10.2.2 Typemap scope</a>	123
<a href="#">10.2.3 Copying a typemap</a>	124

# Table of Contents

<b><u>10 Typemaps</u></b>	
10.2.4 Deleting a typemap	124
10.2.5 Placement of typemaps	124
10.3 Pattern matching rules	125
10.3.1 Basic matching rules	125
10.3.2 Typedef reductions	126
10.3.3 Default typemaps	128
10.3.4 Mixed default typemaps	129
10.3.5 Multi-arguments typemaps	129
10.4 Code generation rules	129
10.4.1 Scope	130
10.4.2 Declaring new local variables	130
10.4.3 Special variables	132
10.5 Common typemap methods	133
10.5.1 "in" typemap	133
10.5.2 "typecheck" typemap	134
10.5.3 "out" typemap	134
10.5.4 "arginit" typemap	134
10.5.5 "default" typemap	135
10.5.6 "check" typemap	135
10.5.7 "argout" typemap	135
10.5.8 "freearg" typemap	135
10.5.9 "newfree" typemap	136
10.5.10 "memberin" typemap	136
10.5.11 "varin" typemap	136
10.5.12 "varout" typemap	136
10.5.13 "throws" typemap	136
10.6 Some typemap examples	137
10.6.1 Typemaps for arrays	137
10.6.2 Implementing constraints with typemaps	140
10.7 Multi-argument typemaps	140
10.8 The run-time type checker	143
10.9 Typemaps and overloading	145
10.10 More about %apply and %clear	149
10.11 Reducing wrapper code size	150
10.12 Passing data between typemaps	151
10.13 Where to go for more information?	151
<b><u>11 Customization Features</u></b>	<b>152</b>
11.1 Exception handling with %exception	152
11.1.1 Handling exceptions in C code	152
11.1.2 Exception handling with longjmp()	153
11.1.3 Handling C++ exceptions	154
11.1.4 Defining different exception handlers	154
11.1.5 Using The SWIG exception library	156
11.2 Object ownership and %newobject	157
11.3 Features and the %feature directive	158
11.3.1 Features and default arguments	159
11.3.2 Feature example	159
<b><u>12 Contracts</u></b>	<b>161</b>
12.1 The %contract directive	161
12.2 %contract and classes	161
12.3 Constant aggregation and %aggregate_check	162

# Table of Contents

<b><u>12 Contracts</u></b>	
<u>12.4 Notes</u>	163
<b><u>13 Variable Length Arguments</u></b>	<b>164</b>
<u>13.1 Introduction</u>	164
<u>13.2 The Problem</u>	165
<u>13.3 Default varargs support</u>	166
<u>13.4 Argument replacement using %varargs</u>	166
<u>13.5 Varargs and typemaps</u>	167
<u>13.6 Varargs wrapping with libffi</u>	168
<u>13.7 Wrapping of va_list</u>	171
<u>13.8 C++ Issues</u>	172
<u>13.9 Discussion</u>	172
<b><u>14 Warning Messages</u></b>	<b>174</b>
<u>14.1 Introduction</u>	174
<u>14.2 Warning message suppression</u>	174
<u>14.3 Enabling additional warnings</u>	175
<u>14.4 Issuing a warning message</u>	175
<u>14.5 Commentary</u>	176
<u>14.6 Warnings as errors</u>	176
<u>14.7 Message output format</u>	176
<u>14.8 Warning number reference</u>	176
<u>14.8.1 Deprecated features (100–199)</u>	176
<u>14.8.2 Preprocessor (200–299)</u>	177
<u>14.8.3 C/C++ Parser (300–399)</u>	177
<u>14.8.4 Types and typemaps (400–499)</u>	178
<u>14.8.5 Code generation (500–599)</u>	178
<u>14.8.6 Language module specific (800–899)</u>	179
<u>14.8.7 User defined (900–999)</u>	179
<u>14.9 History</u>	179
<b><u>15 Working with Modules</u></b>	<b>180</b>
<u>15.1 The SWIG runtime code</u>	180
<u>15.2 A word of caution about static libraries</u>	180
<u>15.3 References</u>	180
<u>15.4 Reducing the wrapper file size</u>	180
<b><u>16 SWIG and C#</u></b>	<b>182</b>
<b><u>17 SWIG and Chicken</u></b>	<b>184</b>
<u>17.1 Preliminaries</u>	184
<u>17.1.1 Running SWIG in C mode</u>	184
<u>17.1.2 Running SWIG in C++ mode</u>	185
<u>17.2 Code Generation</u>	185
<u>17.2.1 Naming Conventions</u>	185
<u>17.2.2 Modules</u>	185
<u>17.2.3 Constants and Variables</u>	185
<u>17.2.4 Functions</u>	185
<u>17.3 TinyCLOS</u>	186
<u>17.4 Compilation</u>	186
<u>17.5 Linkage</u>	186
<u>17.5.1 Shared library</u>	186
<u>17.5.2 Static binary</u>	187

# Table of Contents

<b><u>17 SWIG and Chicken</u></b>	
<u>17.6 Typemaps</u>	187
<u>17.7 Pointers</u>	187
<u>17.8 Unsupported features and known problems</u>	187
<b><u>18 SWIG and Guile</u></b>	<b>188</b>
<u>18.1 Meaning of "Module"</u>	188
<u>18.2 Using the SCM or GH Guile API</u>	188
<u>18.3 Linkage</u>	189
<u>18.3.1 Simple Linkage</u>	189
<u>18.3.2 Passive Linkage</u>	190
<u>18.3.3 Native Guile Module Linkage</u>	190
<u>18.3.4 Old Auto-Loading Guile Module Linkage</u>	190
<u>18.3.5 Hobbit4D Linkage</u>	190
<u>18.4 Underscore Folding</u>	191
<u>18.5 Typemaps</u>	191
<u>18.6 Representation of pointers as smobs</u>	192
<u>18.6.1 GH Smobs</u>	192
<u>18.6.2 SCM Smobs</u>	192
<u>18.6.3 Garbage Collection</u>	192
<u>18.7 Exception Handling</u>	192
<u>18.8 Procedure documentation</u>	193
<u>18.9 Procedures with setters</u>	193
<u>18.10 GOOPS Proxy Classes</u>	193
<u>18.10.1 Naming Issues</u>	195
<u>18.10.2 Linking</u>	196
<b><u>19 SWIG and Java</u></b>	<b>198</b>
<u>19.1 Overview</u>	199
<u>19.2 Preliminaries</u>	200
<u>19.2.1 Running SWIG</u>	200
<u>19.2.2 Additional Commandline Options</u>	200
<u>19.2.3 Getting the right header files</u>	201
<u>19.2.4 Compiling a dynamic module</u>	201
<u>19.2.5 Using your module</u>	201
<u>19.2.6 Dynamic linking problems</u>	202
<u>19.2.7 Compilation problems and compiling with C++</u>	203
<u>19.2.8 Building on Windows</u>	203
<u>19.2.8.1 Running SWIG from Visual Studio</u>	203
<u>19.2.8.2 Using NMAKE</u>	204
<u>19.3 A tour of basic C/C++ wrapping</u>	205
<u>19.3.1 Modules, packages and generated Java classes</u>	205
<u>19.3.2 Functions</u>	205
<u>19.3.3 Global variables</u>	205
<u>19.3.4 Constants</u>	206
<u>19.3.5 Enumerations</u>	208
<u>19.3.5.1 Anonymous enums</u>	208
<u>19.3.5.2 Typesafe enums</u>	208
<u>19.3.5.3 Proper Java enums</u>	209
<u>19.3.5.4 Type unsafe enums</u>	210
<u>19.3.5.5 Simple enums</u>	210
<u>19.3.6 Pointers</u>	211
<u>19.3.7 Structures</u>	211
<u>19.3.8 C++ classes</u>	213



# Table of Contents

## 19 SWIG and Java

19.3.9 C++ inheritance.....	214
19.3.10 Pointers, references, arrays and pass by value.....	214
19.3.10.1 Null pointers.....	215
19.3.11 C++ overloaded functions.....	215
19.3.12 C++ default arguments.....	216
19.3.13 C++ namespaces.....	216
19.3.14 C++ templates.....	217
19.3.15 C++ Smart Pointers.....	217
19.4 Further details on the generated Java classes.....	218
19.4.1 The intermediary JNI class.....	218
19.4.1.1 The intermediary JNI class pragmas.....	219
19.4.2 The Java module class.....	220
19.4.2.1 The Java module class pragmas.....	220
19.4.3 Java proxy classes.....	221
19.4.3.1 Memory management.....	222
19.4.3.2 Inheritance.....	223
19.4.3.3 Proxy classes and garbage collection.....	225
19.4.4 Type wrapper classes.....	226
19.4.5 Enum classes.....	227
19.4.5.1 Typesafe enum classes.....	227
19.4.5.2 Proper Java enum classes.....	228
19.4.5.3 Type unsafe enum classes.....	228
19.5 Cross language polymorphism using directors (experimental).....	229
19.5.1 Enabling directors.....	229
19.5.2 Director classes.....	230
19.5.3 Overhead and code bloat.....	230
19.5.4 Simple directors example.....	230
19.6 Common customization features.....	231
19.6.1 C/C++ helper functions.....	231
19.6.2 Class extension with %extend.....	232
19.6.3 Exception handling with %exception and %javaexception.....	232
19.6.4 Method access with %javamethodmodifiers.....	234
19.7 Tips and techniques.....	234
19.7.1 Input and output parameters using primitive pointers and references.....	234
19.7.2 Simple pointers.....	236
19.7.3 Wrapping C arrays with Java arrays.....	236
19.7.4 Unbounded C Arrays.....	237
19.8 Java typemaps.....	238
19.8.1 Default primitive type mappings.....	239
19.8.2 Sixty four bit JVMs.....	240
19.8.3 What is a typemap?.....	240
19.8.4 Typemaps for mapping C/C++ types to Java types.....	241
19.8.5 Java typemap attributes.....	243
19.8.6 Java special variables.....	243
19.8.7 Typemaps for both C and C++ compilation.....	244
19.8.8 Java code typemaps.....	245
19.8.9 Director specific typemaps.....	247
19.9 Typemap Examples.....	249
19.9.1 Simpler Java enums for enums without initializers.....	249
19.9.2 Handling C++ exception specifications as Java exceptions.....	250
19.9.3 NaN Exception – exception handling for a particular type.....	251
19.9.4 Converting Java String arrays to char **.....	253
19.9.5 Expanding a Java object to multiple arguments.....	254

# Table of Contents

<b><u>19 SWIG and Java</u></b>	
<a href="#">19.9.6 Using typemaps to return arguments</a>	255
<a href="#">19.9.7 Adding Java downcasts to polymorphic return types</a>	257
<a href="#">19.9.8 Adding an equals method to the Java classes</a>	259
<a href="#">19.9.9 Void pointers and a common Java base class</a>	260
<a href="#">19.10 Living with Java Directors</a>	261
<a href="#">19.11 Odds and ends</a>	263
<a href="#">19.11.1 JavaDoc comments</a>	263
<a href="#">19.11.2 Functional interface without proxy classes</a>	263
<a href="#">19.11.3 Using your own JNI functions</a>	264
<a href="#">19.11.4 Performance concerns and hints</a>	264
<a href="#">19.12 Examples</a>	265
<b><u>20 SWIG and Modula-3</u></b>	<b>266</b>
<a href="#">20.1 Overview</a>	266
<a href="#">20.1.1 Why not scripting ?</a>	266
<a href="#">20.1.2 Why Modula-3 ?</a>	267
<a href="#">20.1.3 Why C / C++ ?</a>	267
<a href="#">20.1.4 Why SWIG ?</a>	267
<a href="#">20.2 Conception</a>	267
<a href="#">20.2.1 Interfaces to C libraries</a>	267
<a href="#">20.2.2 Interfaces to C++ libraries</a>	268
<a href="#">20.3 Preliminaries</a>	269
<a href="#">20.3.1 Compilers</a>	269
<a href="#">20.3.2 Additional Commandline Options</a>	269
<a href="#">20.4 Modula-3 typemaps</a>	270
<a href="#">20.4.1 Inputs and outputs</a>	270
<a href="#">20.4.2 Subranges, Enumerations, Sets</a>	271
<a href="#">20.4.3 Objects</a>	271
<a href="#">20.4.4 Imports</a>	271
<a href="#">20.4.5 Exceptions</a>	272
<a href="#">20.4.6 Example</a>	272
<a href="#">20.5 More hints to the generator</a>	272
<a href="#">20.5.1 Features</a>	272
<a href="#">20.5.2 Pragmas</a>	273
<a href="#">20.6 Remarks</a>	273
<b><u>21 SWIG and MzScheme</u></b>	<b>274</b>
<a href="#">21.1 Creating native MzScheme structures</a>	274
<b><u>22 SWIG and Ocaml</u></b>	<b>275</b>
<a href="#">22.1 Preliminaries</a>	275
<a href="#">22.1.1 Running SWIG</a>	276
<a href="#">22.1.2 Compiling the code</a>	276
<a href="#">22.1.3 The camlp4 module</a>	276
<a href="#">22.1.4 Using your module</a>	277
<a href="#">22.1.5 Compilation problems and compiling with C++</a>	277
<a href="#">22.2 The low-level Ocaml/C interface</a>	277
<a href="#">22.2.1 The generated module</a>	278
<a href="#">22.2.2 Enums</a>	279
<a href="#">22.2.2.1 Enum typing in Ocaml</a>	279
<a href="#">22.2.3 Arrays</a>	279
<a href="#">22.2.3.1 Simple types of bounded arrays</a>	279
<a href="#">22.2.3.2 Complex and unbounded arrays</a>	280

# Table of Contents

<b><u>22 SWIG and Ocaml</u></b>	
22.2.3.3 Using an object.....	280
22.2.3.4 Example typemap for a function taking float * and int.....	280
22.2.4 C++ Classes.....	281
22.2.4.1 STL vector and string Example.....	281
22.2.4.2 C++ Class Example.....	282
22.2.4.3 Compiling the example.....	282
22.2.4.4 Sample Session.....	283
22.2.5 Director Classes.....	283
22.2.5.1 Director Introduction.....	283
22.2.5.2 Overriding Methods in Ocaml.....	283
22.2.5.3 Director Usage Example.....	283
22.2.5.4 Creating director objects.....	284
22.2.5.5 Typemaps for directors, directorin, directorout, directorargout.....	285
22.2.5.6 directorin typemap.....	285
22.2.5.7 directorout typemap.....	285
22.2.5.8 directorargout typemap.....	285
22.2.6 Exceptions.....	285
<b><u>23 SWIG and Perl5</u></b> .....	<b>286</b>
23.1 Overview.....	287
23.2 Preliminaries.....	287
23.2.1 Getting the right header files.....	287
23.2.2 Compiling a dynamic module.....	287
23.2.3 Building a dynamic module with MakeMaker.....	288
23.2.4 Building a static version of Perl.....	288
23.2.5 Using the module.....	289
23.2.6 Compilation problems and compiling with C++.....	290
23.2.7 Compiling for 64-bit platforms.....	291
23.3 Building Perl Extensions under Windows.....	292
23.3.1 Running SWIG from Developer Studio.....	292
23.3.2 Using other compilers.....	292
23.4 The low-level interface.....	292
23.4.1 Functions.....	293
23.4.2 Global variables.....	293
23.4.3 Constants.....	293
23.4.4 Pointers.....	294
23.4.5 Structures.....	295
23.4.6 C++ classes.....	296
23.4.7 C++ classes and type-checking.....	297
23.4.8 C++ overloaded functions.....	297
23.4.9 Operators.....	297
23.4.10 Modules and packages.....	298
23.5 Input and output parameters.....	298
23.6 Exception handling.....	300
23.7 Remapping datatypes with typemaps.....	302
23.7.1 A simple typemap example.....	302
23.7.2 Perl5 typemaps.....	303
23.7.3 Typemap variables.....	304
23.7.4 Useful functions.....	305
23.8 Typemap Examples.....	305
23.8.1 Converting a Perl5 array to a char **.....	305
23.8.2 Return values.....	307
23.8.3 Returning values from arguments.....	307

# Table of Contents

<b><u>23 SWIG and Perl5</u></b>	
<u>23.8.4 Accessing array structure members</u>	308
<u>23.8.5 Turning Perl references into C pointers</u>	308
<u>23.8.6 Pointer handling</u>	309
<u>23.9 Proxy classes</u>	310
<u>23.9.1 Preliminaries</u>	310
<u>23.9.2 Structure and class wrappers</u>	310
<u>23.9.3 Object Ownership</u>	312
<u>23.9.4 Nested Objects</u>	312
<u>23.9.5 Proxy Functions</u>	313
<u>23.9.6 Inheritance</u>	313
<u>23.9.7 Modifying the proxy methods</u>	314
<b><u>24 SWIG and PHP4</u></b>	<b>315</b>
<u>24.1 Preliminaries</u>	315
<u>24.2 Building PHP4 Extensions</u>	315
<u>24.2.1 Building a loadable extension</u>	316
<u>24.2.2 Basic PHP4 interface</u>	316
<u>24.2.3 Functions</u>	316
<u>24.2.4 Global Variables</u>	317
<u>24.2.5 Pointers</u>	317
<u>24.2.6 Structures and C++ classes</u>	317
<u>24.2.7 Constants</u>	318
<u>24.2.8 Proxy classes</u>	319
<u>24.2.9 Constructors and Destructors</u>	319
<u>24.2.10 Static Member Variables</u>	319
<u>24.2.11 PHP4 Pragmas</u>	320
<u>24.2.12 Building extensions into php</u>	320
<u>24.2.13 To be furthered...</u>	321
<b><u>25 SWIG and Pike</u></b>	<b>322</b>
<u>25.1 Preliminaries</u>	322
<u>25.1.1 Running SWIG</u>	322
<u>25.1.2 Getting the right header files</u>	322
<u>25.1.3 Using your module</u>	323
<u>25.2 Basic C/C++ Mapping</u>	323
<u>25.2.1 Modules</u>	323
<u>25.2.2 Functions</u>	323
<u>25.2.3 Global variables</u>	323
<u>25.2.4 Constants and enumerated types</u>	324
<u>25.2.5 Constructors and Destructors</u>	324
<u>25.2.6 Static Members</u>	324
<b><u>26 SWIG and Python</u></b>	<b>325</b>
<u>26.1 Overview</u>	326
<u>26.2 Preliminaries</u>	326
<u>26.2.1 Running SWIG</u>	326
<u>26.2.2 Getting the right header files</u>	327
<u>26.2.3 Compiling a dynamic module</u>	327
<u>26.2.4 Using distutils</u>	328
<u>26.2.5 Static linking</u>	328
<u>26.2.6 Using your module</u>	328
<u>26.2.7 Compilation of C++ extensions</u>	330
<u>26.2.8 Compiling for 64-bit platforms</u>	331

# Table of Contents

## **26 SWIG and Python**

26.2.9 Building Python Extensions under Windows.....	331
26.3 A tour of basic C/C++ wrapping.....	332
26.3.1 Modules.....	332
26.3.2 Functions.....	332
26.3.3 Global variables.....	332
26.3.4 Constants and enums.....	333
26.3.5 Pointers.....	334
26.3.6 Structures.....	335
26.3.7 C++ classes.....	337
26.3.8 C++ inheritance.....	338
26.3.9 Pointers, references, values, and arrays.....	338
26.3.10 C++ overloaded functions.....	339
26.3.11 C++ operators.....	340
26.3.12 C++ namespaces.....	340
26.3.13 C++ templates.....	341
26.3.14 C++ Smart Pointers.....	342
26.4 Further details on the Python class interface.....	342
26.4.1 Proxy classes.....	343
26.4.2 Memory management.....	343
26.4.3 Python 2.2 and classic classes.....	345
26.5 Cross language polymorphism (experimental).....	346
26.5.1 Enabling directors.....	346
26.5.2 Director classes.....	347
26.5.3 Ownership and object destruction.....	347
26.5.4 Exception unrolling.....	348
26.5.5 Overhead and code bloat.....	348
26.5.6 Typemaps.....	349
26.5.7 Miscellaneous.....	349
26.6 Common customization features.....	349
26.6.1 C/C++ helper functions.....	349
26.6.2 Adding additional Python code.....	350
26.6.3 Class extension with %extend.....	350
26.6.4 Exception handling with %exception.....	351
26.7 Tips and techniques.....	353
26.7.1 Input and output parameters.....	353
26.7.2 Simple pointers.....	355
26.7.3 Unbounded C Arrays.....	355
26.7.4 String handling.....	356
26.7.5 Arrays.....	356
26.7.6 String arrays.....	357
26.7.7 STL wrappers.....	357
26.8 Typemaps.....	357
26.8.1 What is a typemap?.....	357
26.8.2 Python typemaps.....	358
26.8.3 Typemap variables.....	358
26.8.4 Useful Python Functions.....	359
26.9 Typemap Examples.....	360
26.9.1 Converting Python list to a char **.....	360
26.9.2 Expanding a Python object into multiple arguments.....	361
26.9.3 Using typemaps to return arguments.....	362
26.9.4 Mapping Python tuples into small arrays.....	363
26.9.5 Mapping sequences to C arrays.....	363
26.9.6 Pointer handling.....	364

# Table of Contents

## **26 SWIG and Python**

<u>26.10 Docstring Features</u> .....	365
<u>26.10.1 Module docstring</u> .....	366
<u>26.10.2 %feature("autodoc")</u> .....	366
<u>26.10.2.1 %feature("autodoc", "0")</u> .....	366
<u>26.10.2.2 %feature("autodoc", "1")</u> .....	366
<u>26.10.2.3 %feature("autodoc", "docstring")</u> .....	367
<u>26.10.3 %feature("docstring")</u> .....	367
<u>26.11 Python Packages</u> .....	367

## **27 SWIG and Ruby**.....**368**

<u>27.1 Preliminaries</u> .....	369
<u>27.1.1 Running SWIG</u> .....	369
<u>27.1.2 Getting the right header files</u> .....	369
<u>27.1.3 Compiling a dynamic module</u> .....	369
<u>27.1.4 Using your module</u> .....	370
<u>27.1.5 Static linking</u> .....	370
<u>27.1.6 Compilation of C++ extensions</u> .....	371
<u>27.2 Building Ruby Extensions under Windows 95/NT</u> .....	371
<u>27.2.1 Running SWIG from Developer Studio</u> .....	371
<u>27.3 The Ruby-to-C/C++ Mapping</u> .....	372
<u>27.3.1 Modules</u> .....	372
<u>27.3.2 Functions</u> .....	373
<u>27.3.3 Variable Linking</u> .....	373
<u>27.3.4 Constants</u> .....	374
<u>27.3.5 Pointers</u> .....	374
<u>27.3.6 Structures</u> .....	374
<u>27.3.7 C++ classes</u> .....	376
<u>27.3.8 C++ Inheritance</u> .....	376
<u>27.3.9 C++ Overloaded Functions</u> .....	378
<u>27.3.10 C++ Operators</u> .....	379
<u>27.3.11 C++ namespaces</u> .....	379
<u>27.3.12 C++ templates</u> .....	380
<u>27.3.13 C++ Smart Pointers</u> .....	381
<u>27.3.14 Cross-Language Polymorphism</u> .....	382
<u>27.3.14.1 Exception Unrolling</u> .....	382
<u>27.4 Input and output parameters</u> .....	382
<u>27.5 Simple exception handling</u> .....	384
<u>27.6 Typemaps</u> .....	385
<u>27.6.1 What is a typemap?</u> .....	386
<u>27.6.2 Ruby typemaps</u> .....	387
<u>27.6.3 Typemap variables</u> .....	388
<u>27.6.4 Useful Functions</u> .....	389
<u>27.6.4.1 C Datatypes to Ruby Objects</u> .....	389
<u>27.6.4.2 Ruby Objects to C Datatypes</u> .....	389
<u>27.6.4.3 Macros for VALUE</u> .....	389
<u>27.6.4.4 Exceptions</u> .....	389
<u>27.6.4.5 Iterators</u> .....	390
<u>27.6.5 Typemap Examples</u> .....	391
<u>27.6.6 Converting a Ruby array to a char **</u> .....	391
<u>27.6.7 Collecting arguments in a hash</u> .....	392
<u>27.6.8 Pointer handling</u> .....	395
<u>27.6.8.1 Ruby Datatype Wrapping</u> .....	395
<u>27.7 Operator overloading</u> .....	396

# Table of Contents

## 27 SWIG and Ruby

<u>27.7.1 Example: STL Vector to Ruby Array</u>	396
<u>27.8 Advanced Topics</u>	398
<u>27.8.1 Creating Multi-Module Packages</u>	398
<u>27.8.2 Defining Aliases</u>	399
<u>27.8.3 Predicate Methods</u>	400
<u>27.8.4 Specifying Mixin Modules</u>	400
<u>27.8.5 Interacting with Ruby's Garbage Collector</u>	401

## 28 SWIG and Tcl.....404

<u>28.1 Preliminaries</u>	405
<u>28.1.1 Getting the right header files</u>	405
<u>28.1.2 Compiling a dynamic module</u>	405
<u>28.1.3 Static linking</u>	405
<u>28.1.4 Using your module</u>	406
<u>28.1.5 Compilation of C++ extensions</u>	407
<u>28.1.6 Compiling for 64-bit platforms</u>	408
<u>28.1.7 Setting a package prefix</u>	408
<u>28.1.8 Using namespaces</u>	408
<u>28.2 Building Tcl/Tk Extensions under Windows 95/NT</u>	408
<u>28.2.1 Running SWIG from Developer Studio</u>	409
<u>28.2.2 Using NMAKE</u>	409
<u>28.3 A tour of basic C/C++ wrapping</u>	410
<u>28.3.1 Modules</u>	410
<u>28.3.2 Functions</u>	410
<u>28.3.3 Global variables</u>	411
<u>28.3.4 Constants and enums</u>	411
<u>28.3.5 Pointers</u>	412
<u>28.3.6 Structures</u>	413
<u>28.3.7 C++ classes</u>	415
<u>28.3.8 C++ inheritance</u>	416
<u>28.3.9 Pointers, references, values, and arrays</u>	416
<u>28.3.10 C++ overloaded functions</u>	417
<u>28.3.11 C++ operators</u>	418
<u>28.3.12 C++ namespaces</u>	419
<u>28.3.13 C++ templates</u>	420
<u>28.3.14 C++ Smart Pointers</u>	420
<u>28.4 Further details on the Tcl class interface</u>	421
<u>28.4.1 Proxy classes</u>	421
<u>28.4.2 Memory management</u>	422
<u>28.5 Input and output parameters</u>	423
<u>28.6 Exception handling</u>	425
<u>28.7 Typemaps</u>	427
<u>28.7.1 What is a typemap?</u>	427
<u>28.7.2 Tcl typemaps</u>	428
<u>28.7.3 Typemap variables</u>	429
<u>28.7.4 Converting a Tcl list to a char **</u>	430
<u>28.7.5 Returning values in arguments</u>	431
<u>28.7.6 Useful functions</u>	431
<u>28.7.7 Standard typemaps</u>	432
<u>28.7.8 Pointer handling</u>	432
<u>28.8 Turning a SWIG module into a Tcl Package</u>	433
<u>28.9 Building new kinds of Tcl interfaces (in Tcl)</u>	434
<u>28.9.1 Proxy classes</u>	435

# Table of Contents

<b><u>29 Extending SWIG</u></b>	<b>437</b>
<u>29.1 Introduction</u>	437
<u>29.2 Prerequisites</u>	438
<u>29.3 The Big Picture</u>	438
<u>29.4 Execution Model</u>	438
<u>29.4.1 Preprocessing</u>	439
<u>29.4.2 Parsing</u>	440
<u>29.4.3 Parse Trees</u>	440
<u>29.4.4 Attribute namespaces</u>	444
<u>29.4.5 Symbol Tables</u>	444
<u>29.4.6 The %feature directive</u>	445
<u>29.4.7 Code Generation</u>	446
<u>29.4.8 SWIG and XML</u>	447
<u>29.5 Primitive Data Structures</u>	447
<u>29.5.1 Strings</u>	447
<u>29.5.2 Hashes</u>	449
<u>29.5.3 Lists</u>	449
<u>29.5.4 Common operations</u>	450
<u>29.5.5 Iterating over Lists and Hashes</u>	451
<u>29.5.6 I/O</u>	451
<u>29.6 Navigating and manipulating parse trees</u>	452
<u>29.7 Working with attributes</u>	454
<u>29.8 Type system</u>	455
<u>29.8.1 String encoding of types</u>	455
<u>29.8.2 Type construction</u>	456
<u>29.8.3 Type tests</u>	457
<u>29.8.4 Typedef and inheritance</u>	458
<u>29.8.5 Lvalues</u>	459
<u>29.8.6 Output functions</u>	459
<u>29.9 Parameters</u>	460
<u>29.10 Writing a Language Module</u>	460
<u>29.10.1 Execution model</u>	461
<u>29.10.2 Starting out</u>	461
<u>29.10.3 Command line options</u>	462
<u>29.10.4 Configuration and preprocessing</u>	463
<u>29.10.5 Entry point to code generation</u>	463
<u>29.10.6 Module I/O and wrapper skeleton</u>	464
<u>29.10.7 Low-level code generators</u>	464
<u>29.10.8 Configuration files</u>	464
<u>29.10.9 Runtime support</u>	465
<u>29.10.10 Standard library files</u>	465
<u>29.10.11 Examples and test cases</u>	465
<u>29.10.12 Documentation</u>	466
<u>29.11 Typemaps</u>	466
<u>29.11.1 Proxy classes</u>	466
<u>29.12 Guide to parse tree nodes</u>	466



# SWIG-1.3 Development Documentation

Last update : SWIG-1.3.23 (November 11, 2004)

## Sections

The SWIG documentation is being updated to reflect new SWIG features and enhancements. However, this update process is not quite finished—there is a lot of old SWIG-1.1 documentation and it is taking some time to update all of it. Please pardon our dust (or volunteer to help!).

## SWIG Core Documentation

- [Preface](#)
- [Introduction](#)
- [Getting started on Windows](#)
- [Scripting](#)
- [SWIG Basics](#) (Read this!)
- [SWIG and C++](#)
- [The SWIG preprocessor](#)
- [The SWIG Library](#)
- [Argument handling](#)
- [Typemaps](#)
- [Customization features](#)
- [Contracts](#)
- [Variable length arguments](#)
- [Warning messages](#)
- [Working with Modules](#)

## Language Module Documentation

- [C# support](#)
- [Chicken support](#)
- [Guile support](#)
- [Java support](#)
- [Ocaml support](#)
- [Perl5 support](#)
- [PHP support](#)
- [Python support](#)
- [Ruby support](#)
- [Tcl support](#)

## Developer Documentation

- [Extending SWIG](#)

## Documentation that has not yet been updated

This documentation has not been completely updated from SWIG-1.1, but most of the topics still apply to the current release. Make sure you read the [SWIG Basics](#) chapter before reading any of these chapters. Also, SWIG-1.3.10 features extensive changes to the implementation of typemaps. Make sure you read the [Typemaps](#) chapter above if you are using this feature.

- [Advanced topics](#) (see [Modules](#) for updated information).

# 1 Preface

- [Introduction](#)
- [Special Introduction for Version 1.3](#)
- [SWIG Versions](#)
- [SWIG resources](#)
- [Prerequisites](#)
- [Organization of this manual](#)
- [How to avoid reading the manual](#)
- [Backwards Compatibility](#)
- [Credits](#)
- [Bug reports](#)

## 1.1 Introduction

SWIG (Simplified Wrapper and Interface Generator) is a software development tool for building scripting language interfaces to C and C++ programs. Originally developed in 1995, SWIG was first used by scientists in the Theoretical Physics Division at Los Alamos National Laboratory for building user interfaces to simulation codes running on the Connection Machine 5 supercomputer. In this environment, scientists needed to work with huge amounts of simulation data, complex hardware, and a constantly changing code base. The use of a scripting language interface provided a simple yet highly flexible foundation for solving these types of problems. SWIG simplifies development by largely automating the task of scripting language integration—allowing developers and users to focus on more important problems.

Although SWIG was originally developed for scientific applications, it has since evolved into a general purpose tool that is used in a wide variety of applications—in fact almost anything where C/C++ programming is involved.

## 1.2 Special Introduction for Version 1.3

Since SWIG was released in 1996, its user base and applicability has continued to grow. Although its rate of development has varied, an active development effort has continued to make improvements to the system. Today, nearly a dozen developers are working to create SWIG–2.0—a system that aims to provide wrapping support for nearly all of the ANSI C++ standard and approximately ten target languages including Guile, Java, Mzscheme, Ocaml, Perl, Pike, PHP, Python, Ruby, and Tcl.

## 1.3 SWIG Versions

For several years, the most stable version of SWIG has been release 1.1p5. Starting with version 1.3, a new version numbering scheme has been adopted. Odd version numbers (1.3, 1.5, etc.) represent development versions of SWIG. Even version numbers (1.4, 1.6, etc.) represent stable releases. Currently, developers are working to create a stable SWIG–2.0 release. Don't let the development status of SWIG–1.3 scare you—it is much more stable (and capable) than SWIG–1.1p5.

## 1.4 SWIG resources

The official location of SWIG related material is

<http://www.swig.org>

This site contains the latest version of the software, users guide, and information regarding bugs, installation problems, and implementation tricks.

You can also subscribe to the SWIG mailing list by visiting the page

<http://www.swig.org/mail.html>

The mailing list often discusses some of the more technical aspects of SWIG along with information about beta releases and future work.

CVS access to the latest version of SWIG is also available. More information about this can be obtained at:

<http://www.swig.org/cvs.html>

## 1.5 Prerequisites

This manual assumes that you know how to write C/C++ programs and that you have at least heard of scripting languages such as Tcl, Python, and Perl. A detailed knowledge of these scripting languages is not required although some familiarity won't hurt. No prior experience with building C extensions to these languages is required—after all, this is what SWIG does automatically. However, you should be reasonably familiar with the use of compilers, linkers, and makefiles since making scripting language extensions is somewhat more complicated than writing a normal C program.

Recent SWIG releases have become significantly more capable in their C++ handling—especially support for advanced features like namespaces, overloaded operators, and templates. Whenever possible, this manual tries to cover the technicalities of this interface. However, this isn't meant to be a tutorial on C++ programming. For many of the gory details, you will almost certainly want to consult a good C++ reference. If you don't program in C++, you may just want to skip those parts of the manual.

## 1.6 Organization of this manual

The first few chapters of this manual describe SWIG in general and provide an overview of its capabilities. The remaining chapters are devoted to specific SWIG language modules and are self contained. Thus, if you are using SWIG to build Python interfaces, you can probably skip to that chapter and find almost everything you need to know. Caveat: we are currently working on a documentation rewrite and many of the older language module chapters are still somewhat out of date.

## 1.7 How to avoid reading the manual

If you hate reading manuals, glance at the "Introduction" which contains a few simple examples. These examples contain about 95% of everything you need to know to use SWIG. After that, simply use the language-specific chapters as a reference. The SWIG distribution also comes with a large directory of examples that illustrate different topics.

## 1.8 Backwards Compatibility

If you are a previous user of SWIG, don't expect recent versions of SWIG to provide backwards compatibility. In fact, backwards compatibility issues may arise even between successive 1.3.x releases. Although these incompatibilities are regrettable, SWIG-1.3 is an active development project. The primary goal of this effort is to make SWIG better—a process that would simply be impossible if the developers are constantly bogged down with backwards compatibility issues.

On a positive note, a few incompatibilities are a small price to pay for the large number of new features that have been added—namespaces, templates, smart pointers, overloaded methods, operators, and more.

If you need to work with different versions of SWIG and backwards compatibility is an issue, you can use the `SWIG_VERSION` preprocessor symbol which holds the version of SWIG being executed. `SWIG_VERSION` is a hexadecimal integer such as `0x010311` (corresponding to SWIG-1.3.11). This can be used in an interface file to define different typemaps, take advantage of different features etc:

```
#if SWIG_VERSION >= 0x010311
/* Use some fancy new feature */
#endif
```

Note: The version symbol is not defined in the generated SWIG wrapper file. The SWIG preprocessor has defined `SWIG_VERSION` since SWIG-1.3.11.

## 1.9 Credits

SWIG is an unfunded project that would not be possible without the contributions of many people. Most recent SWIG development has been supported by Matthias Köppe, William Fulton, Lyle Johnson, Richard Palmer, Thien-Thi Nguyen, Jason Stewart, Loic Dachary, Masaki Fukushima, Luigi Ballabio, Sam Liddicott, Art Yerkes, Marcelo Matus, Harco de Hilster, and John Lenz.

Historically, the following people contributed to early versions of SWIG. Peter Lomdahl, Brad Holian, Shujia Zhou, Niels Jensen, and Tim Germann at Los Alamos National Laboratory were the first users. Patrick Tullmann at the University of Utah suggested the idea of automatic documentation generation. John Schmidt and Kurtis Bleeker at the University of Utah tested out the early versions. Chris Johnson supported SWIG's development at the University of Utah. John Buckman, Larry Virden, and Tom Schwaller provided valuable input on the first releases and improving the portability of SWIG. David Fletcher and Gary Holt have provided a great deal of input on improving SWIG's Perl5 implementation. Kevin Butler contributed the first Windows NT port.

## 1.10 Bug reports

Although every attempt has been made to make SWIG bug-free, we are also trying to make feature improvements that may introduce bugs. To report a bug, either send mail to the SWIG developer list at the [swig-dev mailing list](#) or report a bug at the [SWIG bug tracker](#). In your report, be as specific as possible, including (if applicable), error messages, tracebacks (if a core dump occurred), corresponding portions of the SWIG interface file used, and any important pieces of the SWIG generated wrapper code. We can only fix bugs if we know about them.

## 2 Introduction

- [What is SWIG?](#)
- [Why use SWIG?](#)
- [A SWIG example](#)
  - ◆ [SWIG interface file](#)
  - ◆ [The swig command](#)
  - ◆ [Building a Perl5 module](#)
  - ◆ [Building a Python module](#)
  - ◆ [Shortcuts](#)
- [Supported C/C++ language features](#)
- [Non-intrusive interface building](#)
- [Incorporating SWIG into a build system](#)
- [Hands off code generation](#)
- [SWIG and freedom](#)

### 2.1 What is SWIG?

SWIG is a software development tool that simplifies the task of interfacing different languages to C and C++ programs. In a nutshell, SWIG is a compiler that takes C declarations and creates the wrappers needed to access those declarations from other languages including including Perl, Python, Tcl, Ruby, Guile, and Java. SWIG normally requires no modifications to existing code and can often be used to build a usable interface in only a few minutes. Possible applications of SWIG include:

- Building interpreted interfaces to existing C programs.
- Rapid prototyping and application development.
- Interactive debugging.
- Reengineering or refactoring of legacy software into a scripting language components.
- Making a graphical user interface (using Tk for example).
- Testing of C libraries and programs (using scripts).
- Building high performance C modules for scripting languages.
- Making C programming more enjoyable (or tolerable depending on your point of view).
- Impressing your friends.
- Obtaining vast sums of research funding (although obviously not applicable to the author).

SWIG was originally designed to make it extremely easy for scientists and engineers to build extensible scientific software without having to get a degree in software engineering. Because of this, the use of SWIG tends to be somewhat informal and ad-hoc (e.g., SWIG does not require users to provide formal interface specifications as you would find in a dedicated IDL compiler). Although this style of development isn't appropriate for every project, it is particularly well suited to software development in the small; especially the research and development work that is commonly found in scientific and engineering projects.

### 2.2 Why use SWIG?

As stated in the previous section, the primary purpose of SWIG is to simplify the task of integrating C/C++ with other programming languages. However, why would anyone want to do that? To answer that question, it is useful to list a few strengths of C/C++ programming:

- Excellent support for writing programming libraries.
- High performance (number crunching, data processing, graphics, etc.).
- Systems programming and systems integration.
- Large user community and software base.

Next, let's list a few problems with C/C++ programming

- Writing a user interface is rather painful (i.e., consider programming with MFC, X11, GTK, or any number of other libraries).
- Testing is time consuming (the compile/debug cycle).
- Not easy to reconfigure or customize without recompilation.
- Modularization can be tricky.
- Security concerns (buffer overflow for instance).

To address these limitations, many programmers have arrived at the conclusion that it is much easier to use different programming languages for different tasks. For instance, writing a graphical user interface may be significantly easier in a scripting language like Python or Tcl (consider the reasons why millions of programmers have used languages like Visual Basic if you need more proof). An interactive interpreter might also serve as a useful debugging and testing tool. Other languages like Java might greatly simplify the task of writing distributed computing software. The key point is that different programming languages offer different strengths and weaknesses. Moreover, it is extremely unlikely that any programming is ever going to be perfect. Therefore, by combining languages together, you can utilize the best features of each language and greatly simplify certain aspects of software development.

From the standpoint of C/C++, a lot of people use SWIG because they want to break out of the traditional monolithic C programming model which usually results in programs that resemble this:

- A collection of functions and variables that do something useful.
- A `main()` program that starts everything.
- A horrible collection of hacks that form some kind of user interface (but which no-one really wants to touch).

Instead of going down that route, incorporating C/C++ into a higher level language often results in a more modular design, less code, better flexibility, and increased programmer productivity.

SWIG tries to make the problem of C/C++ integration as painless as possible. This allows you to focus on the underlying C program and using the high-level language interface, but not the tedious and complex chore of making the two languages talk to each other. At the same time, SWIG recognizes that all applications are different. Therefore, it provides a wide variety of customization features that let you change almost every aspect of the language bindings. This is the main reason why SWIG has such a large user manual ;-).

## 2.3 A SWIG example

The best way to illustrate SWIG is with a simple example. Consider the following C code:

```
/* File : example.c */

double My_variable = 3.0;

/* Compute factorial of n */
int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

/* Compute n mod m */
int my_mod(int n, int m) {
    return(n % m);
}
```

Suppose that you wanted to access these functions and the global variable `My_variable` from Tcl. You start by making a SWIG interface file as shown below (by convention, these files carry a `.i` suffix) :

### 2.3.1 SWIG interface file

```
/* File : example.i */
%module example
```

```
%{
/* Put headers and other declarations here */
%}

extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
```

The interface file contains ANSI C function prototypes and variable declarations. The `%module` directive defines the name of the module that will be created by SWIG. The `%{ , %}` block provides a location for inserting additional code such as C header files or additional C declarations.

## 2.3.2 The swig command

SWIG is invoked using the `swig` command. We can use this to build a Tcl module (under Linux) as follows :

```
unix > swig -tcl example.i
unix > gcc -c -fpic example.c example_wrap.c -I/usr/local/include
unix > gcc -shared example.o example_wrap.o -o example.so
unix > tclsh
% load ./example.so
% fact 4
24
% my_mod 23 7
2
% expr $My_variable + 4.5
7.5
%
```

The `swig` command produced a new file called `example_wrap.c` that should be compiled along with the `example.c` file. Most operating systems and scripting languages now support dynamic loading of modules. In our example, our Tcl module has been compiled into a shared library that can be loaded into Tcl. When loaded, Tcl can now access the functions and variables declared in the SWIG interface. A look at the file `example_wrap.c` reveals a hideous mess. However, you almost never need to worry about it.

## 2.3.3 Building a Perl5 module

Now, let's turn these functions into a Perl5 module. Without making any changes type the following (shown for Solaris):

```
unix > swig -perl5 example.i
unix > gcc -c example.c example_wrap.c \
-I/usr/local/lib/perl5/sun4-solaris/5.003/CORE
unix > ld -G example.o example_wrap.o -o example.so # This is for Solaris
unix > perl5.003
use example;
print example::fact(4), "\n";
print example::my_mod(23,7), "\n";
print $example::My_variable + 4.5, "\n";
<ctrl-d>
24
2
7.5
unix >
```

## 2.3.4 Building a Python module

Finally, let's build a module for Python (shown for Irix).

```
unix > swig -python example.i
unix > gcc -c -fpic example.c example_wrap.c -I/usr/local/include/python2.0
unix > gcc -shared example.o example_wrap.o -o _example.so
unix > python
```

```

Python 2.0 (#6, Feb 21 2001, 13:29:45)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> import example
>>> example.fact(4)
24
>>> example.my_mod(23,7)
2
>>> example.cvar.My_variable + 4.5
7.5

```

## 2.3.5 Shortcuts

To the truly lazy programmer, one may wonder why we needed the extra interface file at all. As it turns out, you can often do without it. For example, you could also build a Perl5 module by just running SWIG on the C header file and specifying a module name as follows

```

unix > swig -perl5 -module example example.h
unix > gcc -c example.c example_wrap.c \
        -I/usr/local/lib/perl5/sun4-solaris/5.003/CORE
unix > ld -G example.o example_wrap.o -o example.so
unix > perl5.003
use example;
print example::fact(4), "\n";
print example::my_mod(23,7), "\n";
print $example::My_variable + 4.5, "\n";
<ctrl-d>
24
2
7.5

```

## 2.4 Supported C/C++ language features

A primary goal of the SWIG project is to make the language binding process extremely easy. Although a few simple examples have been shown, SWIG is quite capable in supporting most of C++. Some of the major features include:

- Full C99 preprocessing.
- All ANSI C and C++ datatypes.
- Functions, variables, and constants.
- Classes.
- Single and multiple inheritance.
- Overloaded functions and methods.
- Overloaded operators.
- C++ templates (including member templates, specialization, and partial specialization).
- Namespaces.
- Variable length arguments.
- C++ smart pointers.

Currently, the only major C++ feature not supported is nested classes—a limitation that will be removed in a future release.

It is important to stress that SWIG is not a simplistic C++ lexing tool like several apparently similar wrapper generation tools. SWIG not only parses C++, it implements the full C++ type system and it is able to understand C++ semantics. SWIG generates its wrappers with full knowledge of this information. As a result, you will find SWIG to be just as capable of dealing with nasty corner cases as it is in wrapping simple C++ code. In fact, SWIG is able handle C++ code that stresses the very limits of many C++ compilers.



## 2.5 Non-intrusive interface building

When used as intended, SWIG requires minimal (if any) modification to existing C or C++ code. This makes SWIG extremely easy to use with existing packages and promotes software reuse and modularity. By making the C/C++ code independent of the high level interface, you can change the interface and reuse the code in other applications. It is also possible to support different types of interfaces depending on the application.

## 2.6 Incorporating SWIG into a build system

SWIG is a command line tool and as such can be incorporated into any build system that supports invoking external tools/compilers. SWIG is most commonly invoked from within a Makefile, but is also known to be invoked from popular IDEs such as Microsoft Visual Studio.

If you are using the GNU Autotools ([Autoconf](#)/[Automake](#)/[Libtool](#)) to configure SWIG use in your project, the SWIG Autoconf macros can be used. The primary macro is `ac_pkg_swig`, see [http://www.gnu.org/software/ac-archive/htmldoc/ac\\_pkg\\_swig.html](http://www.gnu.org/software/ac-archive/htmldoc/ac_pkg_swig.html). The `ac_python_devel` macro is also helpful for generating Python extensions. See the [Autoconf Macro Archive](#) for further information on this and other Autoconf macros.

There is growing support for SWIG in some build tools, for example [CMake](#) is a cross-platform, open-source build manager with built in support for SWIG. CMake can detect the SWIG executable and many of the target language libraries for linking against. CMake knows how to build shared libraries and loadable modules on many different operating systems. This allows easy cross platform SWIG development. It also can generate the custom commands necessary for driving SWIG from IDE's and makefiles. All of this can be done from a single cross platform input file. The following example is a CMake input file for creating a python wrapper for the SWIG interface file, example.i:

```
# This is a CMake example for Python

FIND_PACKAGE(SWIG REQUIRED)
INCLUDE(${SWIG_USE_FILE})

FIND_PACKAGE(PythonLibs)
INCLUDE_DIRECTORIES(${PYTHON_INCLUDE_PATH})

INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})

SET(CMAKE_SWIG_FLAGS "")

SET_SOURCE_FILES_PROPERTIES(example.i PROPERTIES CPLUSPLUS ON)
SET_SOURCE_FILES_PROPERTIES(example.i PROPERTIES SWIG_FLAGS "-includeall")
SWIG_ADD_MODULE(example python example.i example.cxx)
SWIG_LINK_LIBRARIES(example ${PYTHON_LIBRARIES})
```

The above example will generate native build files such as makefiles, nmake files and Visual Studio projects which will invoke SWIG and compile the generated C++ files into `_example.so` (UNIX) or `_example.dll` (Windows).

## 2.7 Hands off code generation

SWIG is designed to produce working code that needs no hand-modification (in fact, if you look at the output, you probably won't want to modify it). You should think of your target language interface being defined entirely by the input to SWIG, not the resulting output file. While this approach may limit flexibility for hard-core hackers, it allows others to forget about the low-level implementation details.

## 2.8 SWIG and freedom

No, this isn't a special section on the sorry state of world politics. However, it may be useful to know that SWIG was written with

a certain "philosophy" about programming----namely that programmers are smart and that tools should just stay out of their way. Because of that, you will find that SWIG is extremely permissive in what it lets you get away with. In fact, you can use SWIG to go well beyond "shooting yourself in the foot" if dangerous programming is your goal. On the other hand, this kind of freedom may be exactly what is needed to work with complicated and unusual C/C++ applications.

Ironically, the freedom that SWIG provides is countered by an extremely conservative approach to code generation. At it's core, SWIG tries to distill even the most advanced C++ code down to a small well-defined set of interface building techniques based on ANSI C programming. Because of this, you will find that SWIG interfaces can be easily compiled by virtually every C/C++ compiler and that they can be used on any platform. Again, this is an important part of staying out of the programmer's way----the last thing any developer wants to do is to spend their time debugging the output of a tool that relies on non-portable or unreliable programming features.

## 3 Getting started on Windows

- [Installation on Windows](#)
  - ◆ [Windows Executable](#)
- [SWIG Windows Examples](#)
  - ◆ [Instructions for using the Examples with Visual Studio](#)
    - ◇ [Python](#)
    - ◇ [TCL](#)
    - ◇ [Perl](#)
    - ◇ [Java](#)
    - ◇ [Ruby](#)
    - ◇ [C#](#)
  - ◆ [Instructions for using the Examples with other compilers](#)
- [SWIG on Cygwin and MinGW](#)
  - ◆ [Building swig.exe on Windows](#)
    - ◇ [Building swig.exe using MinGW and MSYS](#)
    - ◇ [Building swig.exe using Cygwin](#)
    - ◇ [Building swig.exe alternatives](#)
  - ◆ [Running the examples on Windows using Cygwin](#)

This chapter describes SWIG usage on Microsoft Windows. Installing SWIG and running the examples is covered as well as building the SWIG executable. Usage within the Unix like environments MinGW and Cygwin is also detailed.

### 3.1 Installation on Windows

SWIG does not come with the usual Windows type installation program, however it is quite easy to get started. The main steps are:

- Download the swigwin zip package from the [SWIG website](#) and unzip into a directory. This is all that needs downloading for the Windows platform.
- Set environment variables as described in the [SWIG Windows Examples](#) section in order to run examples using Visual C++.

#### 3.1.1 Windows Executable

The swigwin distribution contains the SWIG Windows executable, swig.exe, which will run on 32 bit versions of Windows, ie Windows 95/98/ME/NT/2000/XP. If you want to build your own swig.exe have a look at [Building swig.exe on Windows](#).

### 3.2 SWIG Windows Examples

Using Microsoft Visual C++ is the most common approach to compiling and linking SWIG's output. The Examples directory has a few Visual C++ project files (.dsp files). These were produced by Visual C++ 6, although they should also work in Visual C++ 5. Later versions of Visual Studio should also be able to open and convert these project files. The C# examples come with .NET 2003 solution (.sln) and project files instead of Visual C++ 6 project files. The project files have been set up to execute SWIG in a custom build rule for the SWIG interface (.i) file. Alternatively run the [examples using Cygwin](#).

More information on each of the examples is available with the examples distributed with SWIG (Examples/index.html).

#### 3.2.1 Instructions for using the Examples with Visual Studio

Ensure the SWIG executable is as supplied in the SWIG root directory in order for the examples to work. Most languages require some environment variables to be set **before** running Visual C++. Note that Visual C++ must be re-started to pick up any changes in environment variables. Open up an example .dsp file, Visual C++ will create a workspace for you (.dsw file). Ensure the Release build is selected then do a Rebuild All from the Build menu. The required environment variables are displayed with their

current values.

The list of required environment variables for each module language is also listed below. They are usually set from the Control Panel and System properties, but this depends on which flavour of Windows you are running. If you don't want to use environment variables then change all occurrences of the environment variables in the .dsp files with hard coded values. If you are interested in how the project files are set up there is explanatory information in some of the language module's documentation.

### 3.2.1.1 Python

**PYTHON\_INCLUDE** : Set this to the directory that contains python.h

**PYTHON\_LIB** : Set this to the python library including path for linking

Example using Python 2.1.1:

PYTHON\_INCLUDE: d:\python21\include

PYTHON\_LIB: d:\python21\libs\python21.lib

### 3.2.1.2 TCL

**TCL\_INCLUDE** : Set this to the directory containing tcl.h

**TCL\_LIB** : Set this to the TCL library including path for linking

Example using ActiveTcl 8.3.3.3

TCL\_INCLUDE: d:\tcl\include

TCL\_LIB: d:\tcl\lib\tcl83.lib

### 3.2.1.3 Perl

**PERL5\_INCLUDE** : Set this to the directory containing perl.h

**PERL5\_LIB** : Set this to the Perl library including path for linking

Example using nsPerl 5.004\_04:

PERL5\_INCLUDE: D:\nsPerl5.004\_04\lib\CORE

PERL5\_LIB: D:\nsPerl5.004\_04\lib\CORE\perl.lib

### 3.2.1.4 Java

**JAVA\_INCLUDE** : Set this to the directory containing jni.h

**JAVA\_BIN** : Set this to the bin directory containing javac.exe

Example using JDK1.3:

JAVA\_INCLUDE: d:\jdk1.3\include

JAVA\_BIN: d:\jdk1.3\bin

### 3.2.1.5 Ruby

**RUBY\_INCLUDE** : Set this to the directory containing ruby.h

**RUBY\_LIB** : Set this to the ruby library including path for linking

Example using Ruby 1.6.4:

RUBY\_INCLUDE: D:\ruby\lib\ruby\1.6\i586-mswin32

RUBY\_LIB: D:\ruby\lib\mswin32-ruby16.lib

### 3.2.1.6 C#

The C# examples do not require any environment variables to be set as a C# project file is included. Just open up the .sln solution file in Visual Studio .NET 2003 and do a Rebuild All from the Build menu. The accompanying C# and C++ project file are automatically used by the solution file.

## 3.2.2 Instructions for using the Examples with other compilers

If you do not have access to Visual C++ you will have to set up project files / Makefiles for your chosen compiler. There is a section in each of the language modules detailing what needs setting up using Visual C++ which may be of some guidance. Alternatively you may want to use Cygwin as described in the following section.

## 3.3 SWIG on Cygwin and MinGW

SWIG can also be compiled and run using [Cygwin](#) or [MinGW](#) which provides a Unix like front end to Windows and comes free with gcc, an ANSI C/C++ compiler. However, this is not a recommended approach as the prebuilt executable is supplied.

### 3.3.1 Building swig.exe on Windows

If you want to replicate the build of swig.exe that comes with the download, follow the MinGW instructions below. This is not necessary to use the supplied swig.exe. This information is provided for those that want to modify the SWIG source code in a Windows environment. Normally this is not needed, so most people will want to ignore this section.

#### 3.3.1.1 Building swig.exe using MinGW and MSYS

- Install MinGW and MSYS from the [MinGW](#) site. This provides a Unix environment on Windows.
- Follow the usual Unix instructions in the README file in the SWIG root directory to build swig.exe from the MinGW command prompt.

#### 3.3.1.2 Building swig.exe using Cygwin

Note that SWIG can also be built using Cygwin. However, the SWIG will then require the Cygwin DLL when executing. Follow the Unix instructions in the README file in the SWIG root directory. Note that the Cygwin environment will also allow one to regenerate the autotool generated files which are supplied with the release distribution. These files are generated using the `autogen.sh` script and will only need regenerating in circumstances such as changing the build system.

#### 3.3.1.3 Building swig.exe alternatives

If you don't want to install Cygwin or MinGW, use a different compiler to build SWIG. For example, all the source code files can be added to a Visual C++ project file in order to build swig.exe from the Visual C++ IDE.

### 3.3.2 Running the examples on Windows using Cygwin

The examples and test-suite work as successfully on Cygwin as on any other Unix operating system. The modules which are known to work are Python, Tcl, Perl, Ruby, Java and C#. Follow the Unix instructions in the README file in the SWIG root directory to build the examples.

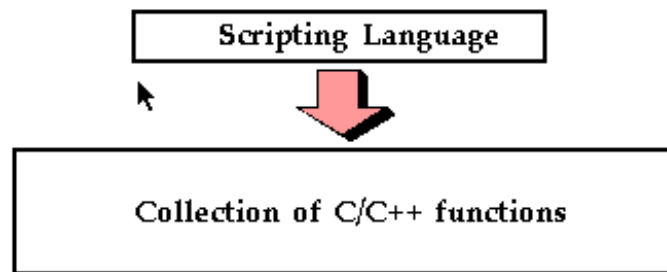
## 4 Scripting Languages

- [The two language view of the world](#)
- [How does a scripting language talk to C?](#)
  - ◆ [Wrapper functions](#)
  - ◆ [Variable linking](#)
  - ◆ [Constants](#)
  - ◆ [Structures and classes](#)
  - ◆ [Proxy classes](#)
- [Building scripting language extensions](#)
  - ◆ [Shared libraries and dynamic loading](#)
  - ◆ [Linking with shared libraries](#)
  - ◆ [Static linking](#)

This chapter provides a brief overview of scripting language extension programming and the mechanisms by which scripting language interpreters access C and C++ code.

### 4.1 The two language view of the world

When a scripting language is used to control a C program, the resulting system tends to look as follows:



In this programming model, the scripting language interpreter is used for high level control whereas the underlying functionality of the C/C++ program is accessed through special scripting language "commands." If you have ever tried to write your own simple command interpreter, you might view the scripting language approach to be a highly advanced implementation of that. Likewise, If you have ever used a package such as MATLAB or IDL, it is a very similar model—the interpreter executes user commands and scripts. However, most of the underlying functionality is written in a low-level language like C or Fortran.

The two-language model of computing is extremely powerful because it exploits the strengths of each language. C/C++ can be used for maximal performance and complicated systems programming tasks. Scripting languages can be used for rapid prototyping, interactive debugging, scripting, and access to high-level data structures such as associative arrays.

### 4.2 How does a scripting language talk to C?

Scripting languages are built around a parser that knows how to execute commands and scripts. Within this parser, there is a mechanism for executing commands and accessing variables. Normally, this is used to implement the builtin features of the language. However, by extending the interpreter, it is usually possible to add new commands and variables. To do this, most languages define a special API for adding new commands. Furthermore, a special foreign function interface defines how these new commands are supposed to hook into the interpreter.

Typically, when you add a new command to a scripting interpreter you need to do two things; first you need to write a special "wrapper" function that serves as the glue between the interpreter and the underlying C function. Then you need to give the interpreter information about the wrapper by providing details about the name of the function, arguments, and so forth. The next few sections illustrate the process.

## 4.2.1 Wrapper functions

Suppose you have an ordinary C function like this :

```
int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}
```

In order to access this function from a scripting language, it is necessary to write a special "wrapper" function that serves as the glue between the scripting language and the underlying C function. A wrapper function must do three things :

- Gather function arguments and make sure they are valid.
- Call the C function.
- Convert the return value into a form recognized by the scripting language.

As an example, the Tcl wrapper function for the `fact()` function above example might look like the following :

```
int wrap_fact(ClientData clientData, Tcl_Interp *interp,
              int argc, char *argv[]) {
    int result;
    int arg0;
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    arg0 = atoi(argv[1]);
    result = fact(arg0);
    sprintf(interp->result, "%d", result);
    return TCL_OK;
}
```

Once you have created a wrapper function, the final step is to tell the scripting language about the new function. This is usually done in an initialization function called by the language when the module is loaded. For example, adding the above function to the Tcl interpreter requires code like the following :

```
int Wrap_Init(Tcl_Interp *interp) {
    Tcl_CreateCommand(interp, "fact", wrap_fact, (ClientData) NULL,
                      (Tcl_CmdDeleteProc *) NULL);
    return TCL_OK;
}
```

When executed, Tcl will now have a new command called "fact" that you can use like any other Tcl command.

Although the process of adding a new function to Tcl has been illustrated, the procedure is almost identical for Perl and Python. Both require special wrappers to be written and both need additional initialization code. Only the specific details are different.

## 4.2.2 Variable linking

Variable linking refers to the problem of mapping a C/C++ global variable to a variable in the scripting language interpreter. For example, suppose you had the following variable:

```
double Foo = 3.5;
```

It might be nice to access it from a script as follows (shown for Perl):

```
$a = $Foo * 2.3;    # Evaluation
$Foo = $a + 2.0;    # Assignment
```

To provide such access, variables are commonly manipulated using a pair of get/set functions. For example, whenever the value of a variable is read, a "get" function is invoked. Similarly, whenever the value of a variable is changed, a "set" function is called.

In many languages, calls to the get/set functions can be attached to evaluation and assignment operators. Therefore, evaluating a variable such as \$Foo might implicitly call the get function. Similarly, typing \$Foo = 4 would call the underlying set function to change the value.

### 4.2.3 Constants

In many cases, a C program or library may define a large collection of constants. For example:

```
#define RED    0xff0000
#define BLUE   0x0000ff
#define GREEN  0x00ff00
```

To make constants available, their values can be stored in scripting language variables such as \$RED, \$BLUE, and \$GREEN. Virtually all scripting languages provide C functions for creating variables so installing constants is usually a trivial exercise.

### 4.2.4 Structures and classes

Although scripting languages have no trouble accessing simple functions and variables, accessing C/C++ structures and classes present a different problem. This is because the implementation of structures is largely related to the problem of data representation and layout. Furthermore, certain language features are difficult to map to an interpreter. For instance, what does C++ inheritance mean in a Perl interface?

The most straightforward technique for handling structures is to implement a collection of accessor functions that hide the underlying representation of a structure. For example,

```
struct Vector {
    Vector();
    ~Vector();
    double x,y,z;
};
```

can be transformed into the following set of functions :

```
Vector *new_Vector();
void delete_Vector(Vector *v);
double Vector_x_get(Vector *v);
double Vector_y_get(Vector *v);
double Vector_z_get(Vector *v);
void Vector_x_set(Vector *v, double x);
void Vector_y_set(Vector *v, double y);
void Vector_z_set(Vector *v, double z);
```

Now, from an interpreter these function might be used as follows:

```
% set v [new_Vector]
% Vector_x_set $v 3.5
% Vector_y_get $v
% delete_Vector $v
% ...
```

Since accessor functions provide a mechanism for accessing the internals of an object, the interpreter does not need to know anything about the actual representation of a Vector.



## 4.2.5 Proxy classes

In certain cases, it is possible to use the low-level accessor functions to create a proxy class, also known as a shadow class. A proxy class is a special kind of object that gets created in a scripting language to access a C/C++ class (or struct) in a way that looks like the original structure (that is, it proxies the real C++ class). For example, if you have the following C definition :

```
class Vector {
public:
    Vector();
    ~Vector();
    double x,y,z;
};
```

A proxy classing mechanism would allow you to access the structure in a more natural manner from the interpreter. For example, in Python, you might want to do this:

```
>>> v = Vector()
>>> v.x = 3
>>> v.y = 4
>>> v.z = -13
>>> ...
>>> del v
```

Similarly, in Perl5 you may want the interface to work like this:

```
$v = new Vector;
$v->{x} = 3;
$v->{y} = 4;
$v->{z} = -13;
```

Finally, in Tcl :

```
Vector v
v configure -x 3 -y 4 -z 13
```

When proxy classes are used, two objects are at really work—one in the scripting language, and an underlying C/C++ object. Operations affect both objects equally and for all practical purposes, it appears as if you are simply manipulating a C/C++ object.

## 4.3 Building scripting language extensions

The final step in using a scripting language with your C/C++ application is adding your extensions to the scripting language itself. There are two primary approaches for doing this. The preferred technique is to build a dynamically loadable extension in the form a shared library. Alternatively, you can recompile the scripting language interpreter with your extensions added to it.

### 4.3.1 Shared libraries and dynamic loading

To create a shared library or DLL, you often need to look at the manual pages for your compiler and linker. However, the procedure for a few common machines is shown below:

```
# Build a shared library for Solaris
gcc -c example.c example_wrap.c -I/usr/local/include
ld -G example.o example_wrap.o -o example.so

# Build a shared library for Linux
agcc -fpic -c example.c example_wrap.c -I/usr/local/include
gcc -shared example.o example_wrap.o -o example.so

# Build a shared library for Irix
```

## SWIG-1.3 Documentation

```
gcc -c example.c example_wrap.c -I/usr/local/include
ld -shared example.o example_wrap.o -o example.so
```

To use your shared library, you simply use the corresponding command in the scripting language (load, import, use, etc...). This will import your module and allow you to start using it. For example:

```
% load ./example.so
% fact 4
24
%
```

When working with C++ codes, the process of building shared libraries may be more complicated—primarily due to the fact that C++ modules may need additional code in order to operate correctly. On many machines, you can build a shared C++ module by following the above procedures, but changing the link line to the following :

```
c++ -shared example.o example_wrap.o -o example.so
```

### 4.3.2 Linking with shared libraries

When building extensions as shared libraries, it is not uncommon for your extension to rely upon other shared libraries on your machine. In order for the extension to work, it needs to be able to find all of these libraries at run-time. Otherwise, you may get an error such as the following :

```
>>> import graph
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "/home/sci/datal/beazley/graph/graph.py", line 2, in ?
    import graphc
ImportError: 1101:/home/sci/datal/beazley/bin/python: rld: Fatal Error: cannot
successfully map soname 'libgraph.so' under any of the filenames /usr/lib/libgraph.so:/
lib/libgraph.so:/lib/cmplrs/cc/libgraph.so:/usr/lib/cmplrs/cc/libgraph.so:
>>>
```

What this error means is that the extension module created by SWIG depends upon a shared library called "libgraph.so" that the system was unable to locate. To fix this problem, there are a few approaches you can take.

- Link your extension and explicitly tell the linker where the required libraries are located. Often times, this can be done with a special linker flag such as `-R`, `-rpath`, etc. This is not implemented in a standard manner so read the man pages for your linker to find out more about how to set the search path for shared libraries.
- Put shared libraries in the same directory as the executable. This technique is sometimes required for correct operation on non-Unix platforms.
- Set the UNIX environment variable `LD_LIBRARY_PATH` to the directory where shared libraries are located before running Python. Although this is an easy solution, it is not recommended. Consider setting the path using linker options instead.

### 4.3.3 Static linking

With static linking, you rebuild the scripting language interpreter with extensions. The process usually involves compiling a short main program that adds your customized commands to the language and starts the interpreter. You then link your program with a library to produce a new scripting language executable.

Although static linking is supported on all platforms, this is not the preferred technique for building scripting language extensions. In fact, there are very few practical reasons for doing this—consider using shared libraries instead.

## 5 SWIG Basics

- [Running SWIG](#)
  - ◆ [Input format](#)
  - ◆ [SWIG Output](#)
  - ◆ [Comments](#)
  - ◆ [C Preprocessor](#)
  - ◆ [SWIG Directives](#)
  - ◆ [Parser Limitations](#)
- [Wrapping Simple C Declarations](#)
  - ◆ [Basic Type Handling](#)
  - ◆ [Global Variables](#)
  - ◆ [Constants](#)
  - ◆ [A brief word about `const`](#)
  - ◆ [A cautionary tale of `char \*`](#)
- [Pointers and complex objects](#)
  - ◆ [Simple pointers](#)
  - ◆ [Run time pointer type checking](#)
  - ◆ [Derived types, structs, and classes](#)
  - ◆ [Undefined datatypes](#)
  - ◆ [Typedef](#)
- [Other Practicalities](#)
  - ◆ [Passing structures by value](#)
  - ◆ [Return by value](#)
  - ◆ [Linking to structure variables](#)
  - ◆ [Linking to `char \*`](#)
  - ◆ [Arrays](#)
  - ◆ [Creating read-only variables](#)
  - ◆ [Renaming and ignoring declarations](#)
  - ◆ [Default/optional arguments](#)
  - ◆ [Pointers to functions and callbacks](#)
- [Structures and unions](#)
  - ◆ [Typedef and structures](#)
  - ◆ [Character strings and structures](#)
  - ◆ [Array members](#)
  - ◆ [Structure data members](#)
  - ◆ [C constructors and destructors](#)
  - ◆ [Adding member functions to C structures](#)
  - ◆ [Nested structures](#)
  - ◆ [Other things to note about structure wrapping](#)
- [Code Insertion](#)
  - ◆ [The output of SWIG](#)
  - ◆ [Code insertion blocks](#)
  - ◆ [Inlined code blocks](#)
  - ◆ [Initialization blocks](#)
- [An Interface Building Strategy](#)
  - ◆ [Preparing a C program for SWIG](#)
  - ◆ [The SWIG interface file](#)
  - ◆ [Why use separate interface files?](#)
  - ◆ [Getting the right header files](#)
  - ◆ [What to do with `main\(\)`](#)

This chapter describes the basic operation of SWIG, the structure of its input files, and how it handles standard ANSI C declarations. C++ support is described in the next chapter. However, C++ programmers should still read this chapter to understand the basics. Specific details about each target language are described in later chapters.

## 5.1 Running SWIG

To run SWIG, use the `swig` command with one or more of the following options and a filename like this:

```
swig [ options ] filename

-chicken          Generate CHICKEN wrappers
-csharp           Generate C# wrappers
-guile            Generate Guile wrappers
-java             Generate Java wrappers
-mzscheme         Generate Mzscheme wrappers
-ocaml            Generate Ocaml wrappers
-perl             Generate Perl wrappers
-php             Generate PHP wrappers
-pike             Generate Pike wrappers
-python           Generate Python wrappers
-ruby             Generate Ruby wrappers
-sexp             Generate Lisp S-Expressions wrappers
-tcl              Generate Tcl wrappers
-xml              Generate XML wrappers
-c++              Enable C++ parsing
-Dsymbol          Define a preprocessor symbol
-Fstandard        Display error/warning messages in commonly used format
-Fmicrosoft       Display error/warning messages in Microsoft format
-help            Display all options
-Idir             Add a directory to the file include path
-lfile            Include a SWIG library file.
-module name      Set the name of the SWIG module
-o outfile        Name of output file
-outdir dir       Set language specific files output directory
-swiglib          Show location of SWIG library
-version          Show SWIG version number
```

This is a subset of commandline options. Additional options are also defined for each target language. A full list can be obtained by typing `swig -help` or `swig -lang -help`.

### 5.1.1 Input format

As input, SWIG expects a file containing ANSI C/C++ declarations and special SWIG directives. More often than not, this is a special SWIG interface file which is usually denoted with a special `.i` or `.swg` suffix. In certain cases, SWIG can be used directly on raw header files or source files. However, this is not the most typical case and there are several reasons why you might not want to do this (described later).

The most common format of a SWIG interface is as follows:

```
%module mymodule
%{
#include "myheader.h"
%}
// Now list ANSI C/C++ declarations
int foo;
int bar(int x);
...
```

The name of the module is supplied using the special `%module` directive (or the `-module` command line option). This directive must appear at the beginning of the file and is used to name the resulting extension module (in addition, this name often defines a namespace in the target language). If the module name is supplied on the command line, it overrides the name specified with the `%module` directive.

Everything in the `%{ ... %}` block is simply copied verbatim to the resulting wrapper file created by SWIG. This section is almost always used to include header files and other declarations that are required to make the generated wrapper code compile. It

is important to emphasize that just because you include a declaration in a SWIG input file, that declaration does *not* automatically appear in the generated wrapper code—therefore you need to make sure you include the proper header files in the `%{ ... %}` section. It should be noted that the text enclosed in `%{ ... %}` is not parsed or interpreted by SWIG. The `%{ ... %}` syntax and semantics in SWIG is analogous to that of the declarations section used in input files to parser generation tools such as yacc or bison.

### 5.1.2 SWIG Output

The output of SWIG is a C/C++ file that contains all of the wrapper code needed to build an extension module. SWIG may generate some additional files depending on the target language. By default, an input file with the name `file.i` is transformed into a file `file_wrap.c` or `file_wrap.cxx` (depending on whether or not the `-c++` option has been used). The name of the output file can be changed using the `-o` option. In certain cases, file suffixes are used by the compiler to determine the source language (C, C++, etc.). Therefore, you have to use the `-o` option to change the suffix of the SWIG-generated wrapper file if you want something different than the default. For example:

```
$ swig -c++ -python -o example_wrap.cpp example.i
```

The C/C++ output file created by SWIG often contains everything that is needed to construct a extension module for the target scripting language. SWIG is not a stub compiler nor is it usually necessary to edit the output file (and if you look at the output, you probably won't want to). To build the final extension module, the SWIG output file is compiled and linked with the rest of your C/C++ program to create a shared library.

Many target languages will also generate proxy class files in the target language. The default output directory for these language specific files is the same directory as the generated C/C++ file. This can be modified using the `-outdir` option. For example:

```
$ swig -c++ -python -outdir pyfiles -o cppfiles/example_wrap.cpp example.i
```

If the directories `cppfiles` and `pyfiles` exist, the following will be generated:

```
cppfiles/example_wrap.cpp
pyfiles/example.py
```

### 5.1.3 Comments

C and C++ style comments may appear anywhere in interface files. In previous versions of SWIG, comments were used to generate documentation files. However, this feature is currently under repair and will reappear in a later SWIG release.

### 5.1.4 C Preprocessor

Like C, SWIG preprocesses all input files through an enhanced version of the C preprocessor. All standard preprocessor features are supported including file inclusion, conditional compilation and macros. However, `#include` statements are ignored unless the `-includeall` command line option has been supplied. The reason for disabling includes is that SWIG is sometimes used to process raw C header files. In this case, you usually only want the extension module to include functions in the supplied header file rather than everything that might be included by that header file (i.e., system headers, C library functions, etc.).

It should also be noted that the SWIG preprocessor skips all text enclosed inside a `%{ ... %}` block. In addition, the preprocessor includes a number of macro handling enhancements that make it more powerful than the normal C preprocessor. These extensions are described in the "[Preprocessor](#)" chapter.

### 5.1.5 SWIG Directives

Most of SWIG's operation is controlled by special directives that are always preceded by a `"%"` to distinguish them from normal C declarations. These directives are used to give SWIG hints or to alter SWIG's parsing behavior in some manner.

Since SWIG directives are not legal C syntax, it is generally not possible to include them in header files. However, SWIG directives can be included in C header files using conditional compilation like this:

```
/* header.h --- Some header file */

/* SWIG directives -- only seen if SWIG is running */
#ifdef SWIG
%module foo
#endif
```

SWIG is a special preprocessing symbol defined by SWIG when it is parsing an input file.

### 5.1.6 Parser Limitations

Although SWIG can parse most C/C++ declarations, it does not provide a complete C/C++ parser implementation. Most of these limitations pertain to very complicated type declarations and certain advanced C++ features. Specifically, the following features are not currently supported:

- Non-conventional type declarations. For example, SWIG does not support declarations such as the following (even though this is legal C):

```
/* Non-conventional placement of storage specifier (extern) */
const int extern Number;

/* Extra declarator grouping */
Matrix (foo);    // A global variable

/* Extra declarator grouping in parameters */
void bar(Spam (Grok)(Doh));
```

In practice, few (if any) C programmers actually write code like this since this style is never featured in programming books. However, if you're feeling particularly obfuscated, you can certainly break SWIG (although why would you want to?).

- Running SWIG on C++ source files (what would appear in a .C or .cxx file) is not recommended. Even though SWIG can parse C++ class declarations, it ignores declarations that are decoupled from their original class definition (the declarations are parsed, but a lot of warning messages may be generated). For example:

```
/* Not supported by SWIG */
int foo::bar(int) {
    ... whatever ...
}
```

- Certain advanced features of C++ such as nested classes are not yet supported. Please see the section on using SWIG with C++ for more information.

In the event of a parsing error, conditional compilation can be used to skip offending code. For example:

```
#ifndef SWIG
... some bad declarations ...
#endif
```

Alternatively, you can just delete the offending code from the interface file.

One of the reasons why SWIG does not provide a full C++ parser implementation is that it has been designed to work with incomplete specifications and to be very permissive in its handling of C/C++ datatypes (e.g., SWIG can generate interfaces even when there are missing class declarations or opaque datatypes). Unfortunately, this approach makes it extremely difficult to implement certain parts of a C/C++ parser as most compilers use type information to assist in the parsing of more complex declarations (for the truly curious, the primary complication in the implementation is that the SWIG parser does not utilize a separate *typedef-name* terminal symbol as described on p. 234 of K&R).

## 5.2 Wrapping Simple C Declarations

SWIG wraps simple C declarations by creating an interface that closely matches the way in which the declarations would be used in a C program. For example, consider the following interface file:

```
%module example

extern double sin(double x);
extern int strcmp(const char *, const char *);
extern int Foo;
#define STATUS 50
#define VERSION "1.1"
```

In this file, there are two functions `sin()` and `strcmp()`, a global variable `Foo`, and two constants `STATUS` and `VERSION`. When SWIG creates an extension module, these declarations are accessible as scripting language functions, variables, and constants respectively. For example, in Tcl:

```
% sin 3
5.2335956
% strcmp Dave Mike
-1
% puts $Foo
42
% puts $STATUS
50
% puts $VERSION
1.1
```

Or in Python:

```
>>> example.sin(3)
5.2335956
>>> example strcmp('Dave', 'Mike')
-1
>>> print example.cvar.Foo
42
>>> print example.STATUS
50
>>> print example.VERSION
1.1
```

Whenever possible, SWIG creates an interface that closely matches the underlying C/C++ code. However, due to subtle differences between languages, run-time environments, and semantics, it is not always possible to do so. The next few sections describes various aspects of this mapping.

### 5.2.1 Basic Type Handling

In order to build an interface, SWIG has to convert C/C++ datatypes to equivalent types in the target language. Generally, scripting languages provide a more limited set of primitive types than C. Therefore, this conversion process involves a certain amount of type coercion.

Most scripting languages provide a single integer type that is implemented using the `int` or `long` datatype in C. The following list shows all of the C datatypes that SWIG will convert to and from integers in the target language:

```
int
short
long
unsigned
signed
unsigned short
unsigned long
```

```
unsigned char
signed char
bool
```

When an integral value is converted from C, a cast is used to convert it to the representation in the target language. Thus, a 16 bit short in C may be promoted to a 32 bit integer. When integers are converted in the other direction, the value is cast back into the original C type. If the value is too large to fit, it is silently truncated.

`unsigned char` and `signed char` are special cases that are handled as small 8-bit integers. Normally, the `char` datatype is mapped as a one-character ASCII string.

The `bool` datatype is cast to and from an integer value of 0 and 1 unless the target language provides a special boolean type.

Some care is required when working with large integer values. Most scripting languages use 32-bit integers so mapping a 64-bit long integer may lead to truncation errors. Similar problems may arise with 32 bit unsigned integers (which may appear as large negative numbers). As a rule of thumb, the `int` datatype and all variations of `char` and `short` datatypes are safe to use. For `unsigned int` and `long` datatypes, you will need to carefully check the correct operation of your program after it has been wrapped with SWIG.

Although the SWIG parser supports the `long long` datatype, not all language modules support it. This is because `long long` usually exceeds the integer precision available in the target language. In certain modules such as Tcl and Perl5, `long long` integers are encoded as strings. This allows the full range of these numbers to be represented. However, it does not allow `long long` values to be used in arithmetic expressions. It should also be noted that although `long long` is part of the ISO C99 standard, it is not universally supported by all C compilers. Make sure you are using a compiler that supports `long long` before trying to use this type with SWIG.

SWIG recognizes the following floating point types :

```
float
double
```

Floating point numbers are mapped to and from the natural representation of floats in the target language. This is almost always a C `double`. The rarely used datatype of `long double` is not supported by SWIG.

The `char` datatype is mapped into a NULL terminated ASCII string with a single character. When used in a scripting language it shows up as a tiny string containing the character value. When converting the value back into C, SWIG takes a character string from the scripting language and strips off the first character as the `char` value. Thus if the value "foo" is assigned to a `char` datatype, it gets the value 'f'.

The `char *` datatype is handled as a NULL-terminated ASCII string. SWIG maps this into a 8-bit character string in the target scripting language. SWIG converts character strings in the target language to NULL terminated strings before passing them into C/C++. The default handling of these strings does not allow them to have embedded NULL bytes. Therefore, the `char *` datatype is not generally suitable for passing binary data. However, it is possible to change this behavior by defining a SWIG typemap. See the chapter on [Typemaps](#) for details about this.

At this time, SWIG does not provide any special support for Unicode or wide-character strings (the C `wchar_t` type). This is a delicate topic that is poorly understood by many programmers and not implemented in a consistent manner across languages. For those scripting languages that provide Unicode support, Unicode strings are often available in an 8-bit representation such as UTF-8 that can be mapped to the `char *` type (in which case the SWIG interface will probably work). If the program you are wrapping uses Unicode, there is no guarantee that Unicode characters in the target language will use the same internal representation (e.g., UCS-2 vs. UCS-4). You may need to write some special conversion functions.

## 5.2.2 Global Variables

Whenever possible, SWIG maps C/C++ global variables into scripting language variables. For example,

```
%module example
```

### 5.2.2 Global Variables



```
double foo;
```

results in a scripting language variable like this:

```
# Tcl
set foo [3.5]           ;# Set foo to 3.5
puts $foo               ;# Print the value of foo

# Python
cvar.foo = 3.5          # Set foo to 3.5
print cvar.foo          # Print value of foo

# Perl
$foo = 3.5;             # Set foo to 3.5
print $foo, "\n";       # Print value of foo

# Ruby
Module.foo = 3.5        # Set foo to 3.5
print Module.foo, "\n"  # Print value of foo
```

Whenever the scripting language variable is used, the underlying C global variable is accessed. Although SWIG makes every attempt to make global variables work like scripting language variables, it is not always possible to do so. For instance, in Python, all global variables must be accessed through a special variable object known as `cvar` (shown above). In Ruby, variables are accessed as attributes of the module. Other languages may convert variables to a pair of accessor functions. For example, the Java module generates a pair of functions `double get_foo()` and `set_foo(double val)` that are used to manipulate the value.

Finally, if a global variable has been declared as `const`, it only supports read-only access. Note: this behavior is new to SWIG-1.3. Earlier versions of SWIG incorrectly handled `const` and created constants instead.

### 5.2.3 Constants

Constants can be created using `#define`, enumerations, or a special `%constant` directive. The following interface file shows a few valid constant declarations :

```
#define I_CONST          5                // An integer constant
#define PI               3.14159         // A Floating point constant
#define S_CONST          "hello world"   // A string constant
#define NEWLINE          '\n'            // Character constant

enum boolean {NO=0, YES=1};
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
             SEP, OCT, NOV, DEC};
%constant double BLAH = 42.37;
#define F_CONST (double) 5               // A floating pointer constant with cast
#define PI_4 PI/4
#define FLAGS 0x04 | 0x08 | 0x40
```

In `#define` declarations, the type of a constant is inferred by syntax. For example, a number with a decimal point is assumed to be floating point. In addition, SWIG must be able to fully resolve all of the symbols used in a `#define` in order for a constant to actually be created. This restriction is necessary because `#define` is also used to define preprocessor macros that are definitely not meant to be part of the scripting language interface. For example:

```
#define EXTERN extern

EXTERN void foo();
```

In this case, you probably don't want to create a constant called `EXTERN` (what would the value be?). In general, SWIG will not create constants for macros unless the value can be completely determined by the preprocessor. For instance, in the above

example, the declaration

```
#define PI_4 PI/4
```

defines a constant because `PI` was already defined as a constant and the value is known.

The use of constant expressions is allowed, but SWIG does not evaluate them. Rather, it passes them through to the output file and lets the C compiler perform the final evaluation (SWIG does perform a limited form of type-checking however).

For enumerations, it is critical that the original enum definition be included somewhere in the interface file (either in a header file or in the `%{ , %}` block). SWIG only translates the enumeration into code needed to add the constants to a scripting language. It needs the original enumeration declaration in order to get the correct enum values as assigned by the C compiler.

The `%constant` directive is used to more precisely create constants corresponding to different C datatypes. Although it is not usually not needed for simple values, it is more useful when working with pointers and other more complex datatypes. Typically, `%constant` is only used when you want to add constants to the scripting language interface that are not defined in the original header file.

## 5.2.4 A brief word about `const`

A common confusion with C programming is the semantic meaning of the `const` qualifier in declarations—especially when it is mixed with pointers and other type modifiers. In fact, previous versions of SWIG handled `const` incorrectly—a situation that SWIG-1.3.7 and newer releases have fixed.

Starting with SWIG-1.3, all variable declarations, regardless of any use of `const`, are wrapped as global variables. If a declaration happens to be declared as `const`, it is wrapped as a read-only variable. To tell if a variable is `const` or not, you need to look at the right-most occurrence of the `const` qualifier (that appears before the variable name). If the right-most `const` occurs after all other type modifiers (such as pointers), then the variable is `const`. Otherwise, it is not.

Here are some examples of `const` declarations.

```
const char a;           // A constant character
char const b;           // A constant character (the same)
char *const c;          // A constant pointer to a character
const char *const d;    // A constant pointer to a constant character
```

Here is an example of a declaration that is not `const`:

```
const char *e;          // A pointer to a constant character. The pointer
                        // may be modified.
```

In this case, the pointer `e` can change—it's only the value being pointed to that is read-only.

**Compatibility Note:** One reason for changing SWIG to handle `const` declarations as read-only variables is that there are many situations where the value of a `const` variable might change. For example, a library might export a symbol as `const` in its public API to discourage modification, but still allow the value to change through some other kind of internal mechanism. Furthermore, programmers often overlook the fact that with a constant declaration like `char *const`, the underlying data being pointed to can be modified—it's only the pointer itself that is constant. In an embedded system, a `const` declaration might refer to a read-only memory address such as the location of a memory-mapped I/O device port (where the value changes, but writing to the port is not supported by the hardware). Rather than trying to build a bunch of special cases into the `const` qualifier, the new interpretation of `const` as "read-only" is simple and exactly matches the actual semantics of `const` in C/C++. If you really want to create a constant as in older versions of SWIG, use the `%constant` directive instead. For example:

```
%constant double PI = 3.14159;
```

or

```
#ifndef SWIG
```

```
#define const %constant
#endif
const double foo = 3.4;
const double bar = 23.4;
const int    spam = 42;
#ifdef SWIG
#undef const
#endif
...
```

### 5.2.5 A cautionary tale of `char *`

Before going any further, there is one bit of caution involving `char *` that must now be mentioned. When strings are passed from a scripting language to a C `char *`, the pointer usually points to string data stored inside the interpreter. It is almost always a really bad idea to modify this data. Furthermore, some languages may explicitly disallow it. For instance, in Python, strings are supposed to be immutable. If you violate this, you will probably receive a vast amount of wrath when you unleash your module on the world.

The primary source of problems are functions that might modify string data in place. A classic example would be a function like this:

```
char *strcat(char *s, const char *t)
```

Although SWIG will certainly generate a wrapper for this, its behavior will be undefined. In fact, it will probably cause your application to crash with a segmentation fault or other memory related problem. This is because `s` refers to some internal data in the target language—data that you shouldn't be touching.

The bottom line: don't rely on `char *` for anything other than read-only input values. However, it must be noted that you could change the behavior of SWIG using [typemaps](#).

## 5.3 Pointers and complex objects

Most C programs manipulate arrays, structures, and other types of objects. This section discusses the handling of these datatypes.

### 5.3.1 Simple pointers

Pointers to primitive C datatypes such as

```
int *
double ***
char **
```

are fully supported by SWIG. Rather than trying to convert the data being pointed to into a scripting representation, SWIG simply encodes the pointer itself into a representation that contains the actual value of the pointer and a type-tag. Thus, the SWIG representation of the above pointers (in Tcl), might look like this:

```
_10081012_p_int
_1008e124_ppp_double
_f8ac_pp_char
```

A NULL pointer is represented by the string "NULL" or the value 0 encoded with type information.

All pointers are treated as opaque objects by SWIG. Thus, a pointer may be returned by a function and passed around to other C functions as needed. For all practical purposes, the scripting language interface works in exactly the same way as you would use the pointer in a C program. The only difference is that there is no mechanism for dereferencing the pointer since this would require the target language to understand the memory layout of the underlying object.

The scripting language representation of a pointer value should never be manipulated directly. Even though the values shown look like hexadecimal addresses, the numbers used may differ from the actual machine address (e.g., on little-endian machines, the digits may appear in reverse order). Furthermore, SWIG does not normally map pointers into high-level objects such as associative arrays or lists (for example, converting an `int *` into an list of integers). There are several reasons why SWIG does not do this:

- There is not enough information in a C declaration to properly map pointers into higher level constructs. For example, an `int *` may indeed be an array of integers, but if it contains ten million elements, converting it into a list object is probably a bad idea.
- The underlying semantics associated with a pointer is not known by SWIG. For instance, an `int *` might not be an array at all—perhaps it is an output value!
- By handling all pointers in a consistent manner, the implementation of SWIG is greatly simplified and less prone to error.

### 5.3.2 Run time pointer type checking

By allowing pointers to be manipulated from a scripting language, extension modules effectively bypass compile-time type checking in the C/C++ compiler. To prevent errors, a type signature is encoded into all pointer values and is used to perform run-time type checking. This type-checking process is an integral part of SWIG and can not be disabled or modified without using `typemaps` (described in later chapters).

Like C, `void *` matches any kind of pointer. Furthermore, `NULL` pointers can be passed to any function that expects to receive a pointer. Although this has the potential to cause a crash, `NULL` pointers are also sometimes used as sentinel values or to denote a missing/empty value. Therefore, SWIG leaves `NULL` pointer checking up to the application.

### 5.3.3 Derived types, structs, and classes

For everything else (structs, classes, arrays, etc...) SWIG applies a very simple rule :

#### Everything else is a pointer

In other words, SWIG manipulates everything else by reference. This model makes sense because most C/C++ programs make heavy use of pointers and SWIG can use the type-checked pointer mechanism already present for handling pointers to basic datatypes.

Although this probably sounds complicated, it's really quite simple. Suppose you have an interface file like this :

```
%module fileio
FILE *fopen(char *, char *);
int fclose(FILE *);
unsigned fread(void *ptr, unsigned size, unsigned nobj, FILE *);
unsigned fwrite(void *ptr, unsigned size, unsigned nobj, FILE *);
void *malloc(int nbytes);
void free(void *);
```

In this file, SWIG doesn't know what a `FILE` is, but since it's used as a pointer, so it doesn't really matter what it is. If you wrapped this module into Python, you can use the functions just like you expect :

```
# Copy a file
def filecopy(source,target):
    f1 = fopen(source,"r")
    f2 = fopen(target,"w")
    buffer = malloc(8192)
    nbytes = fread(buffer,8192,1,f1)
    while (nbytes > 0):
        fwrite(buffer,8192,1,f2)
        nbytes = fread(buffer,8192,1,f1)
    free(buffer)
```

In this case `f1`, `f2`, and `buffer` are all opaque objects containing C pointers. It doesn't matter what value they contain—our program works just fine without this knowledge.

### 5.3.4 Undefined datatypes

When SWIG encounters an undeclared datatype, it automatically assumes that it is a structure or class. For example, suppose the following function appeared in a SWIG input file:

```
void matrix_multiply(Matrix *a, Matrix *b, Matrix *c);
```

SWIG has no idea what a "Matrix" is. However, it is obviously a pointer to something so SWIG generates a wrapper using its generic pointer handling code.

Unlike C or C++, SWIG does not actually care whether `Matrix` has been previously defined in the interface file or not. This allows SWIG to generate interfaces from only partial or limited information. In some cases, you may not care what a `Matrix` really is as long as you can pass an opaque reference to one around in the scripting language interface.

An important detail to mention is that SWIG will gladly generate wrappers for an interface when there are unspecified type names. However, **all unspecified types are internally handled as pointers to structures or classes!** For example, consider the following declaration:

```
void foo(size_t num);
```

If `size_t` is undeclared, SWIG generates wrappers that expect to receive a type of `size_t *` (this mapping is described shortly). As a result, the scripting interface might behave strangely. For example:

```
foo(40);
TypeError: expected a _p_size_t.
```

The only way to fix this problem is to make sure you properly declare type names using `typedef`.

### 5.3.5 Typedef

Like C, `typedef` can be used to define new type names in SWIG. For example:

```
typedef unsigned int size_t;
```

`typedef` definitions appearing in a SWIG interface are not propagated to the generated wrapper code. Therefore, they either need to be defined in an included header file or placed in the declarations section like this:

```
%{
/* Include in the generated wrapper file */
typedef unsigned int size_t;
%}
/* Tell SWIG about it */
typedef unsigned int size_t;
```

or

```
%inline %{
typedef unsigned int size_t;
%}
```

In certain cases, you might be able to include other header files to collect type information. For example:

```
%module example
%import "sys/types.h"
```

In this case, you might run SWIG as follows:

```
$ swig -I/usr/include -includeall example.i
```

It should be noted that your mileage will vary greatly here. System headers are notoriously complicated and may rely upon a variety of non-standard C coding extensions (e.g., such as special directives to GCC). Unless you exactly specify the right include directories and preprocessor symbols, this may not work correctly (you will have to experiment).

SWIG tracks `typedef` declarations and uses this information for run-time type checking. For instance, if you use the above `typedef` and had the following function declaration:

```
void foo(unsigned int *ptr);
```

The corresponding wrapper function will accept arguments of type `unsigned int *` or `size_t *`.

## 5.4 Other Practicalities

So far, this chapter has presented almost everything you need to know to use SWIG for simple interfaces. However, some C programs use idioms that are somewhat more difficult to map to a scripting language interface. This section describes some of these issues.

### 5.4.1 Passing structures by value

Sometimes a C function takes structure parameters that are passed by value. For example, consider the following function:

```
double dot_product(Vector a, Vector b);
```

To deal with this, SWIG transforms the function to use pointers by creating a wrapper equivalent to the following:

```
double wrap_dot_product(Vector *a, Vector *b) {
    Vector x = *a;
    Vector y = *b;
    return dot_product(x,y);
}
```

In the target language, the `dot_product()` function now accepts pointers to Vectors instead of Vectors. For the most part, this transformation is transparent so you might not notice.

### 5.4.2 Return by value

C functions that return structures or classes datatypes by value are more difficult to handle. Consider the following function:

```
Vector cross_product(Vector v1, Vector v2);
```

This function wants to return `Vector`, but SWIG only really supports pointers. As a result, SWIG creates a wrapper like this:

```
Vector *wrap_cross_product(Vector *v1, Vector *v2) {
    Vector x = *v1;
    Vector y = *v2;
    Vector *result;
    result = (Vector *) malloc(sizeof(Vector));
    *(result) = cross(x,y);
    return result;
}
```

or if SWIG was run with the `-c++` option:

```
Vector *wrap_cross(Vector *v1, Vector *v2) {
    Vector x = *v1;
```

```

    Vector y = *v2;
    Vector *result = new Vector(cross(x,y)); // Uses default copy constructor
    return result;
}

```

In both cases, SWIG allocates a new object and returns a reference to it. It is up to the user to delete the returned object when it is no longer in use. Clearly, this will leak memory if you are unaware of the implicit memory allocation and don't take steps to free the result. That said, it should be noted that some language modules can now automatically track newly created objects and reclaim memory for you. Consult the documentation for each language module for more details.

It should also be noted that the handling of pass/return by value in C++ has some special cases. For example, the above code fragments don't work correctly if `Vector` doesn't define a default constructor. The section on SWIG and C++ has more information about this case.

### 5.4.3 Linking to structure variables

When global variables or class members involving structures are encountered, SWIG handles them as pointers. For example, a global variable like this

```
Vector unit_i;
```

gets mapped to an underlying pair of set/get functions like this :

```

Vector *unit_i_get() {
    return &unit_i;
}
void unit_i_set(Vector *value) {
    unit_i = *value;
}

```

Again some caution is in order. A global variable created in this manner will show up as a pointer in the target scripting language. It would be an extremely bad idea to free or destroy such a pointer. Also, C++ classes must supply a properly defined copy constructor in order for assignment to work correctly.

### 5.4.4 Linking to char \*

When a global variable of type `char *` appears, SWIG uses `malloc()` or `new` to allocate memory for the new value. Specifically, if you have a variable like this

```
char *foo;
```

SWIG generates the following code:

```

/* C mode */
void foo_set(char *value) {
    if (foo) free(foo);
    foo = (char *) malloc(strlen(value)+1);
    strcpy(foo,value);
}

/* C++ mode.  When -c++ option is used */
void foo_set(char *value) {
    if (foo) delete [] foo;
    foo = new char[strlen(value)+1];
    strcpy(foo,value);
}

```

If this is not the behavior that you want, consider making the variable read-only using the `%immutable` directive. Alternatively, you might write a short assist-function to set the value exactly like you want. For example:

```
%inline %{
    void set_foo(char *value) {
        strncpy(foo,value, 50);
    }
%}
```

Note: If you write an assist function like this, you will have to call it as a function from the target scripting language (it does not work like a variable). For example, in Python you will have to write:

```
>>> set_foo("Hello World")
```

A common mistake with `char *` variables is to link to a variable declared like this:

```
char *VERSION = "1.0";
```

In this case, the variable will be readable, but any attempt to change the value results in a segmentation or general protection fault. This is due to the fact that SWIG is trying to release the old value using `free` or `delete` when the string literal value currently assigned to the variable wasn't allocated using `malloc()` or `new`. To fix this behavior, you can either mark the variable as read-only, write a typemap (as described in Chapter 6), or write a special set function as shown. Another alternative is to declare the variable as an array:

```
char VERSION[64] = "1.0";
```

When variables of type `const char *` are declared, SWIG still generates functions for setting and getting the value. However, the default behavior does *not* release the previous contents (resulting in a possible memory leak). In fact, you may get a warning message such as this when wrapping such a variable:

```
example.i:20. Typemap warning. Setting const char * variable may leak memory
```

The reason for this behavior is that `const char *` variables are often used to point to string literals. For example:

```
const char *foo = "Hello World\n";
```

Therefore, it's a really bad idea to call `free()` on such a pointer. On the other hand, it *is* legal to change the pointer to point to some other value. When setting a variable of this type, SWIG allocates a new string (using `malloc` or `new`) and changes the pointer to point to the new value. However, repeated modifications of the value will result in a memory leak since the old value is not released.

## 5.4.5 Arrays

Arrays are fully supported by SWIG, but they are always handled as pointers instead of mapping them to a special array object or list in the target language. Thus, the following declarations :

```
int foobar(int a[40]);
void grok(char *argv[]);
void transpose(double a[20][20]);
```

are processed as if they were really declared like this:

```
int foobar(int *a);
void grok(char **argv);
void transpose(double (*a)[20]);
```

Like C, SWIG does not perform array bounds checking. It is up to the user to make sure the pointer points a suitably allocated region of memory.

Multi-dimensional arrays are transformed into a pointer to an array of one less dimension. For example:

```
int [10];           // Maps to int *
```



```
int [10][20];      // Maps to int (*)[20]
int [10][20][30]; // Maps to int (*)[20][30]
```

It is important to note that in the C type system, a multidimensional array `a[ ][ ]` is **NOT** equivalent to a single pointer `*a` or a double pointer such as `**a`. Instead, a pointer to an array is used (as shown above) where the actual value of the pointer is the starting memory location of the array. The reader is strongly advised to dust off their C book and re-read the section on arrays before using them with SWIG.

Array variables are supported, but are read-only by default. For example:

```
int    a[100][200];
```

In this case, reading the variable 'a' returns a pointer of type `int (*)[200]` that points to the first element of the array `&a[0][0]`. Trying to modify 'a' results in an error. This is because SWIG does not know how to copy data from the target language into the array. To work around this limitation, you may want to write a few simple assist functions like this:

```
%inline %{
void a_set(int i, int j, int val) {
    a[i][j] = val;
}
int a_get(int i, int j) {
    return a[i][j];
}
%}
```

To dynamically create arrays of various sizes and shapes, it may be useful to write some helper functions in your interface. For example:

```
// Some array helpers
%inline %{
/* Create any sort of [size] array */
int *int_array(int size) {
    return (int *) malloc(size*sizeof(int));
}
/* Create a two-dimension array [size][10] */
int (*int_array_10(int size))[10] {
    return (int (*)[10]) malloc(size*10*sizeof(int));
}
%}
```

Arrays of char are handled as a special case by SWIG. In this case, strings in the target language can be stored in the array. For example, if you have a declaration like this,

```
char pathname[256];
```

SWIG generates functions for both getting and setting the value that are equivalent to the following code:

```
char *pathname_get() {
    return pathname;
}
void pathname_set(char *value) {
    strncpy(pathname,value,256);
}
```

In the target language, the value can be set like a normal variable.

### 5.4.6 Creating read-only variables

A read-only variable can be created by using the `%immutable` directive as shown :

```
// File : interface.i
```

#### 5.4.6 Creating read-only variables

```

int      a;                // Can read/write
%immutable;
int      b,c,d             // Read only variables
%mutable;
double   x,y               // read/write

```

The `%immutable` directive enables read-only mode until it is explicitly disabled using the `%mutable` directive. As an alternative to turning read-only mode off and on like this, individual declarations can also be tagged as immutable. For example:

```

%immutable x;              // Make x read-only
...
double x;                  // Read-only (from earlier %immutable directive)
double y;                  // Read-write
...

```

Read-only variables are also created when declarations are declared as `const`. For example:

```

const int foo;              /* Read only variable */
char * const version="1.0"; /* Read only variable */

```

**Compatibility note:** Read-only access used to be controlled by a pair of directives `%readonly` and `%readwrite`. Although these directives still work, they generate a warning message. Simply change the directives to `%immutable`; and `%mutable`; to silence the warning. Don't forget the extra semicolon!

## 5.4.7 Renaming and ignoring declarations

Normally, the name of a C declaration is used when that declaration is wrapped into the target language. However, this may generate a conflict with a keyword or already existing function in the scripting language. To resolve a name conflict, you can use the `%rename` directive as shown :

```

// interface.i

%rename(my_print) print;
extern void print(char *);

%rename(foo) a_really_long_and_annoying_name;
extern int a_really_long_and_annoying_name;

```

SWIG still calls the correct C function, but in this case the function `print()` will really be called `"my_print()"` in the target language.

The placement of the `%rename` directive is arbitrary as long as it appears before the declarations to be renamed. A common technique is to write code for wrapping a header file like this:

```

// interface.i

%rename(my_print) print;
%rename(foo) a_really_long_and_annoying_name;

#include "header.h"

```

`%rename` applies a renaming operation to all future occurrences of a name. The renaming applies to functions, variables, class and structure names, member functions, and member data. For example, if you had two-dozen C++ classes, all with a member function named `'print'` (which is a keyword in Python), you could rename them all to `'output'` by specifying :

```

%rename(output) print; // Rename all 'print' functions to 'output'

```

SWIG does not normally perform any checks to see if the functions it wraps are already defined in the target scripting language. However, if you are careful about namespaces and your use of modules, you can usually avoid these problems.

Closely related to `%rename` is the `%ignore` directive. `%ignore` instructs SWIG to ignore declarations that match a given identifier. For example:

```
%ignore print;           // Ignore all declarations named print
%ignore _HAVE_FOO_H;     // Ignore an include guard constant
...
#include "foo.h"          // Grab a header file
...
```

One use of `%ignore` is to selectively remove certain declarations from a header file without having to add conditional compilation to the header. However, it should be stressed that this only works for simple declarations. If you need to remove a whole section of problematic code, the SWIG preprocessor should be used instead.

More powerful variants of `%rename` and `%ignore` directives can be used to help wrap C++ overloaded functions and methods or C++ methods which use default arguments. This is described in the [Ambiguity resolution and renaming](#) section in the C++ chapter.

**Compatibility note:** Older versions of SWIG provided a special `%name` directive for renaming declarations. For example:

```
%name(output) extern void print(char *);
```

This directive is still supported, but it is deprecated and should probably be avoided. The `%rename` directive is more powerful and better supports wrapping of raw header file information.

## 5.4.8 Default/optional arguments

SWIG supports default arguments in both C and C++ code. For example:

```
int plot(double x, double y, int color=WHITE);
```

In this case, SWIG generates wrapper code where the default arguments are optional in the target language. For example, this function could be used in Tcl as follows :

```
% plot -3.4 7.5                # Use default value
% plot -3.4 7.5 10             # set color to 10 instead
```

Although the ANSI C standard does not allow default arguments, default arguments specified in a SWIG interface work with both C and C++.

**Note:** There is a subtle semantic issue concerning the use of default arguments and the SWIG generated wrapper code. When default arguments are used in C code, the default values are emitted into the wrappers and the function is invoked with a full set of arguments. This is different to when wrapping C++ where an overloaded wrapper method is generated for each defaulted argument. Please refer to the section on [default arguments](#) in the C++ chapter for further details.

## 5.4.9 Pointers to functions and callbacks

Occasionally, a C library may include functions that expect to receive pointers to functions—possibly to serve as callbacks. SWIG provides full support for function pointers provided that the callback functions are defined in C and not in the target language. For example, consider a function like this:

```
int binary_op(int a, int b, int (*op)(int,int));
```

When you first wrap something like this into an extension module, you may find the function to be impossible to use. For instance, in Python:

```
>>> def add(x,y):
...     return x+y
```

```
...
>>> binary_op(3,4,add)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Type error. Expected _p_f_int_int__int
>>>
```

The reason for this error is that SWIG doesn't know how to map a scripting language function into a C callback. However, existing C functions can be used as arguments provided you install them as constants. One way to do this is to use the `%constant` directive like this:

```
/* Function with a callback */
int binary_op(int a, int b, int (*op)(int,int));

/* Some callback functions */
%constant int add(int,int);
%constant int sub(int,int);
%constant int mul(int,int);
```

In this case, `add`, `sub`, and `mul` become function pointer constants in the target scripting language. This allows you to use them as follows:

```
>>> binary_op(3,4,add)
7
>>> binary_op(3,4,mul)
12
>>>
```

Unfortunately, by declaring the callback functions as constants, they are no longer accessible as functions. For example:

```
>>> add(3,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object is not callable: '_ff020efc_p_f_int_int__int'
>>>
```

If you want to make a function available as both a callback function and a function, you can use the `%callback` and `%nocallback` directives like this:

```
/* Function with a callback */
int binary_op(int a, int b, int (*op)(int,int));

/* Some callback functions */
%callback("%s_cb")
int add(int,int);
int sub(int,int);
int mul(int,int);
%nocallback
```

The argument to `%callback` is a printf-style format string that specifies the naming convention for the callback constants (`%s` gets replaced by the function name). The callback mode remains in effect until it is explicitly disabled using `%nocallback`. When you do this, the interface now works as follows:

```
>>> binary_op(3,4,add_cb)
7
>>> binary_op(3,4,mul_cb)
12
>>> add(3,4)
7
>>> mul(3,4)
12
```

Notice that when the function is used as a callback, special names such as `add_cb` is used instead. To call the function normally, just use the original function name such as `add()`.

SWIG provides a number of extensions to standard C `printf` formatting that may be useful in this context. For instance, the following variation installs the callbacks as all upper-case constants such as `ADD`, `SUB`, and `MUL`:

```
/* Some callback functions */
%callback("%(upper)s")
int add(int,int);
int sub(int,int);
int mul(int,int);
%nocallback
```

A format string of `"%(lower)s"` converts all characters to lower-case. A string of `"%(title)s"` capitalizes the first character and converts the rest to lower case.

And now, a final note about function pointer support. Although SWIG does not normally allow callback functions to be written in the target language, this can be accomplished with the use of typemaps and other advanced SWIG features. This is described in a later chapter.

## 5.5 Structures and unions

This section describes the behavior of SWIG when processing ANSI C structures and union declarations. Extensions to handle C++ are described in the next section.

If SWIG encounters the definition of a structure or union, it creates a set of accessor functions. Although SWIG does not need structure definitions to build an interface, providing definitions make it possible to access structure members. The accessor functions generated by SWIG simply take a pointer to an object and allow access to an individual member. For example, the declaration :

```
struct Vector {
    double x,y,z;
}
```

gets transformed into the following set of accessor functions :

```
double Vector_x_get(struct Vector *obj) {
    return obj->x;
}
double Vector_y_get(struct Vector *obj) {
    return obj->y;
}
double Vector_z_get(struct Vector *obj) {
    return obj->z;
}
void Vector_x_set(struct Vector *obj, double value) {
    obj->x = value;
}
void Vector_y_set(struct Vector *obj, double value) {
    obj->y = value;
}
void Vector_z_set(struct Vector *obj, double value) {
    obj->z = value;
}
```

In addition, SWIG creates default constructor and destructor functions if none are defined in the interface. For example:

```
struct Vector *new_Vector() {
    return (Vector *) calloc(1,sizeof(struct Vector));
}
```

```
void delete_Vector(struct Vector *obj) {
    free(obj);
}
```

Using these low-level accessor functions, an object can be minimally manipulated from the target language using code like this:

```
v = new_Vector()
Vector_x_set(v,2)
Vector_y_set(v,10)
Vector_z_set(v,-5)
...
delete_Vector(v)
```

However, most of SWIG's language modules also provide a high-level interface that is more convenient. Keep reading.

### 5.5.1 Typedef and structures

SWIG supports the following construct which is quite common in C programs :

```
typedef struct {
    double x,y,z;
} Vector;
```

When encountered, SWIG assumes that the name of the object is `Vector' and creates accessor functions like before. The only difference is that the use of typedef allows SWIG to drop the struct keyword on its generated code. For example:

```
double Vector_x_get(Vector *obj) {
    return obj->x;
}
```

If two different names are used like this :

```
typedef struct vector_struct {
    double x,y,z;
} Vector;
```

the name Vector is used instead of vector\_struct since this is more typical C programming style. If declarations defined later in the interface use the type struct vector\_struct, SWIG knows that this is the same as Vector and it generates the appropriate type-checking code.

### 5.5.2 Character strings and structures

Structures involving character strings require some care. SWIG assumes that all members of type char \* have been dynamically allocated using malloc() and that they are NULL-terminated ASCII strings. When such a member is modified, the previously contents will be released, and the new contents allocated. For example :

```
%module mymodule
...
struct Foo {
    char *name;
    ...
}
```

This results in the following accessor functions :

```
char *Foo_name_get(Foo *obj) {
    return Foo->name;
}
```

```
char *Foo_name_set(Foo *obj, char *c) {
    if (obj->name) free(obj->name);
    obj->name = (char *) malloc(strlen(c)+1);
    strcpy(obj->name, c);
    return obj->name;
}
```

If this behavior differs from what you need in your applications, the SWIG "memberin" typemap can be used to change it. See the typemaps chapter for further details.

Note: If the `-c++` option is used, `new` and `delete` are used to perform memory allocation.

## 5.5.3 Array members

Arrays may appear as the members of structures, but they will be read-only. SWIG will write an accessor function that returns the pointer to the first element of the array, but will not write a function to change the contents of the array itself. When this situation is detected, SWIG may generate a warning message such as the following :

```
interface.i:116. Warning. Array member will be read-only
```

To eliminate the warning message, typemaps can be used, but this is discussed in a later chapter. In many cases, the warning message is harmless.

## 5.5.4 Structure data members

Occasionally, a structure will contain data members that are themselves structures. For example:

```
typedef struct Foo {
    int x;
} Foo;

typedef struct Bar {
    int y;
    Foo f;          /* struct member */
} Bar;
```

When a structure member is wrapped, it is always handled as a pointer. For example:

```
Foo *Bar_f_get(Bar *b) {
    return &b->f;
}
void Bar_f_set(Bar *b, Foo *value) {
    b->f = *value;
}
```

The reasons for this are somewhat subtle but have to do with the problem of modifying and accessing data inside the data member. For example, suppose you wanted to modify the value of `f.x` of a `Bar` object like this:

```
Bar *b;
b->f.x = 37;
```

Translating this assignment to function calls (as would be used inside the scripting language interface) results in the following code:

```
Bar *b;
Foo_x_set(Bar_f_get(b), 37);
```

In this code, if the `Bar_f_get()` function were to return a `Foo` instead of a `Foo *`, then the resulting modification would be applied to a *copy* of `f` and not the data member `f` itself. Clearly that's not what you want!

It should be noted that this transformation to pointers only occurs if SWIG knows that a data member is a structure or class. For instance, if you had a structure like this,

```
struct Foo {
    WORD    w;
};
```

and nothing was known about WORD, then SWIG will generate more normal accessor functions like this:

```
WORD Foo_w_get(Foo *f) {
    return f->w;
}
void Foo_w_set(Foo *f, WORD value) {
    f->w = value;
}
```

**Compatibility Note:** SWIG-1.3.11 and earlier releases transformed all non-primitive member datatypes to pointers. Starting in SWIG-1.3.12, this transformation *only* occurs if a datatype is known to be a structure, class, or union. This is unlikely to break existing code. However, if you need to tell SWIG that an undeclared datatype is really a struct, simply use a forward struct declaration such as "struct Foo;".

## 5.5.5 C constructors and destructors

When wrapping structures, it is generally useful to have a mechanism for creating and destroying objects. If you don't do anything, SWIG will automatically generate functions for creating and destroying objects using `malloc()` and `free()`. Note: the use of `malloc()` only applies when SWIG is used on C code (i.e., when the `-c++` option is *not* supplied on the command line). C++ is handled differently.

If you don't want SWIG to generate constructors and destructors, you can use the `%nodefault` directive or the `-no_default` command line option. For example:

```
swig -no_default example.i
```

or

```
%module foo
...
%nodefault;          // Don't create default constructors/destructors
... declarations ...
%makedefault;        // Reenable default constructors/destructors
```

If you need more precise control, `%nodefault` can selectively target individual structure definitions. For example:

```
%nodefault Foo;      // No default constructor/destructors for Foo
...
struct Foo {          // No default generated.
};

struct Bar {          // Default constructor/destructor generated.
};
```

**Compatibility note:** Prior to SWIG-1.3.7, SWIG did not generate default constructors or destructors unless you explicitly turned them on using `-make_default`. However, it appears that most users want to have constructor and destructor functions so it has now been enabled as the default behavior.

## 5.5.6 Adding member functions to C structures

Most languages provide a mechanism for creating classes and supporting object oriented programming. From a C standpoint, object oriented programming really just boils down to the process of attaching functions to structures. These functions normally



operate on an instance of the structure (or object). Although there is a natural mapping of C++ to such a scheme, there is no direct mechanism for utilizing it with C code. However, SWIG provides a special `%extend` directive that makes it possible to attach methods to C structures for purposes of building an object oriented interface. Suppose you have a C header file with the following declaration :

```
/* file : vector.h */
...
typedef struct {
    double x,y,z;
} Vector;
```

You can make a `Vector` look alot like a class by writing a SWIG interface like this:

```
// file : vector.i
%module mymodule
%{
#include "vector.h"
%}

#include vector.h          // Just grab original C header file
%extend Vector {          // Attach these functions to struct Vector
    Vector(double x, double y, double z) {
        Vector *v;
        v = (Vector *v) malloc(sizeof(Vector));
        v->x = x;
        v->y = y;
        v->z = z;
        return v;
    }
    ~Vector() {
        free(self);
    }
    double magnitude() {
        return sqrt(self->x*self->x+self->y*self->y+self->z*self->z);
    }
    void print() {
        printf("Vector [%g, %g, %g]\n", self->x,self->y,self->z);
    }
};
```

Now, when used with proxy classes in Python, you can do things like this :

```
>>> v = Vector(3,4,0)          # Create a new vector
>>> print v.magnitude()       # Print magnitude
5.0
>>> v.print()                 # Print it out
[ 3, 4, 0 ]
>>> del v                     # Destroy it
```

The `%extend` directive can also be used inside the definition of the `Vector` structure. For example:

```
// file : vector.i
%module mymodule
%{
#include "vector.h"
%}

typedef struct {
    double x,y,z;
    %extend {
        Vector(double x, double y, double z) { ... }
        ~Vector() { ... }
        ...
    }
};
```

```
    }
} Vector;
```

Finally, `%extend` can be used to access externally written functions provided they follow the naming convention used in this example :

```
/* File : vector.c */
/* Vector methods */
#include "vector.h"
Vector *new_Vector(double x, double y, double z) {
    Vector *v;
    v = (Vector *) malloc(sizeof(Vector));
    v->x = x;
    v->y = y;
    v->z = z;
    return v;
}
void delete_Vector(Vector *v) {
    free(v);
}

double Vector_magnitude(Vector *v) {
    return sqrt(v->x*v->x+v->y*v->y+v->z*v->z);
}

// File : vector.i
// Interface file
%module mymodule
%{
#include "vector.h"
}%

typedef struct {
    double x,y,z;
    %extend {
        Vector(int,int,int); // This calls new_Vector()
        ~Vector();           // This calls delete_Vector()
        double magnitude();  // This will call Vector_magnitude()
        ...
    }
} Vector;
```

A little known feature of the `%extend` directive is that it can also be used to add synthesized attributes or to modify the behavior of existing data attributes. For example, suppose you wanted to make `magnitude` a read-only attribute of `Vector` instead of a method. To do this, you might write some code like this:

```
// Add a new attribute to Vector
%extend Vector {
    const double magnitude;
}
// Now supply the implementation of the Vector_magnitude_get function
%{
const double Vector_magnitude_get(Vector *v) {
    return (const double) return sqrt(v->x*v->x+v->y*v->y+v->z*v->z);
}
}%
```

Now, for all practical purposes, `magnitude` will appear like an attribute of the object.

A similar technique can also be used to work with problematic data members. For example, consider this interface:

```
struct Person {
    char name[50];
    ...
}
```

```
}
```

By default, the name attribute is read-only because SWIG does not normally know how to modify arrays. However, you can rewrite the interface as follows to change this:

```
struct Person {
    %extend {
        char *name;
    }
    ...
}

// Specific implementation of set/get functions
%{
char *Person_name_get(Person *p) {
    return p->name;
}
void Person_name_set(Person *p, char *val) {
    strncpy(p->name, val, 50);
}
%}
```

Finally, it should be stressed that even though `%extend` can be used to add new data members, these new members can not require the allocation of additional storage in the object (e.g., their values must be entirely synthesized from existing attributes of the structure).

**Compatibility note:** The `%extend` directive is a new name for the `%addmethods` directive. Since `%addmethods` could be used to extend a structure with more than just methods, a more suitable directive name has been chosen.

## 5.5.7 Nested structures

Occasionally, a C program will involve structures like this :

```
typedef struct Object {
    int objtype;
    union {
        int      ivalue;
        double   dvalue;
        char     *strvalue;
        void     *ptrvalue;
    } intRep;
} Object;
```

When SWIG encounters this, it performs a structure splitting operation that transforms the declaration into the equivalent of the following:

```
typedef union {
    int      ivalue;
    double   dvalue;
    char     *strvalue;
    void     *ptrvalue;
} Object_intRep;

typedef struct Object {
    int objType;
    Object_intRep intRep;
} Object;
```

SWIG will then create an `Object_intRep` structure for use inside the interface file. Accessor functions will be created for both structures. In this case, functions like this would be created :

```

Object_intRep *Object_intRep_get(Object *o) {
    return (Object_intRep *) &o->intRep;
}
int Object_intRep_ivalue_get(Object_intRep *o) {
    return o->ivalue;
}
int Object_intRep_ivalue_set(Object_intRep *o, int value) {
    return (o->ivalue = value);
}
double Object_intRep_dvalue_get(Object_intRep *o) {
    return o->dvalue;
}
... etc ...

```

Although this process is a little hairy, it works like you would expect in the target scripting language—especially when proxy classes are used. For instance, in Perl:

```

# Perl5 script for accessing nested member
$o = CreateObject();                # Create an object somehow
$o->{intRep}->{ivalue} = 7          # Change value of o.intRep.ivalue

```

If you have a lot nested structure declarations, it is advisable to double-check them after running SWIG. Although, there is a good chance that they will work, you may have to modify the interface file in certain cases.

### 5.5.8 Other things to note about structure wrapping

SWIG doesn't care if the declaration of a structure in a `.i` file exactly matches that used in the underlying C code (except in the case of nested structures). For this reason, there are no problems omitting problematic members or simply omitting the structure definition altogether. If you are happy passing pointers around, this can be done without ever giving SWIG a structure definition.

Starting with SWIG1.3, a number of improvements have been made to SWIG's code generator. Specifically, even though structure access has been described in terms of high-level accessor functions such as this,

```

double Vector_x_get(Vector *v) {
    return v->x;
}

```

most of the generated code is actually inlined directly into wrapper functions. Therefore, no function `Vector_x_get()` actually exists in the generated wrapper file. For example, when creating a Tcl module, the following function is generated instead:

```

static int
_wrap_Vector_x_get(ClientData clientData, Tcl_Interp *interp,
    int objc, Tcl_Obj *CONST objv[]) {
    struct Vector *arg1 ;
    double result ;

    if (SWIG_GetArgs(interp, objc, objv, "p:Vector_x_get self ", &arg0,
        SWIGTYPE_p_Vector) == TCL_ERROR)
        return TCL_ERROR;
    result = (double) (arg1->x);
    Tcl_SetObjResult(interp, Tcl_NewDoubleObj((double) result));
    return TCL_OK;
}

```

The only exception to this rule are methods defined with `%extend`. In this case, the added code is contained in a separate function.

Finally, it is important to note that most language modules may choose to build a more advanced interface. Although you may never use the low-level interface described here, most of SWIG's language modules use it in some way or another.

## 5.6 Code Insertion

Sometimes it is necessary to insert special code into the resulting wrapper file generated by SWIG. For example, you may want to include additional C code to perform initialization or other operations. There are four common ways to insert code, but it's useful to know how the output of SWIG is structured first.

### 5.6.1 The output of SWIG

When SWIG creates its output file, it is broken up into four sections corresponding to runtime code, headers, wrapper functions, and module initialization code (in that order).

- **Runtime code.**

This code is internal to SWIG and is used to include type-checking and other support functions that are used by the rest of the module.

- **Header section.**

This is user-defined support code that has been included by the `%{ ... %}` directive. Usually this consists of header files and other helper functions.

- **Wrapper code.**

These are the wrappers generated automatically by SWIG.

- **Module initialization.**

The function generated by SWIG to initialize the module upon loading.

### 5.6.2 Code insertion blocks

Code is inserted into the appropriate code section by using one of the following code insertion directives:

```
%runtime %{
    ... code in runtime section ...
}%

%header %{
    ... code in header section ...
}%

%wrapper %{
    ... code in wrapper section ...
}%

%init %{
    ... code in init section ...
}%
```

The bare `%{ ... %}` directive is a shortcut that is the same as `%header %{ ... %}`.

Everything in a code insertion block is copied verbatim into the output file and is not parsed by SWIG. Most SWIG input files have at least one such block to include header files and support C code. Additional code blocks may be placed anywhere in a SWIG file as needed.

```
%module mymodule
%{
#include "my_header.h"
}%
... Declare functions here
%{

void some_extra_function() {
    ...
}
}%
```

A common use for code blocks is to write "helper" functions. These are functions that are used specifically for the purpose of building an interface, but which are generally not visible to the normal C program. For example :

```
%{
/* Create a new vector */
static Vector *new_Vector() {
    return (Vector *) malloc(sizeof(Vector));
}

%}
// Now wrap it
Vector *new_Vector();
```

## 5.6.3 Inlined code blocks

Since the process of writing helper functions is fairly common, there is a special inlined form of code block that is used as follows :

```
%inline %{
/* Create a new vector */
Vector *new_Vector() {
    return (Vector *) malloc(sizeof(Vector));
}
%}
```

The `%inline` directive inserts all of the code that follows verbatim into the header portion of an interface file. The code is then parsed by both the SWIG preprocessor and parser. Thus, the above example creates a new command `new_Vector` using only one declaration. Since the code inside an `%inline` `%{ ... %}` block is given to both the C compiler and SWIG, it is illegal to include any SWIG directives inside a `%{ ... %}` block.

## 5.6.4 Initialization blocks

When code is included in the `%init` section, it is copied directly into the module initialization function. For example, if you needed to perform some extra initialization on module loading, you could write this:

```
%init %{
    init_variables();
%}
```

## 5.7 An Interface Building Strategy

This section describes the general approach for building interface with SWIG. The specifics related to a particular scripting language are found in later chapters.

### 5.7.1 Preparing a C program for SWIG

SWIG doesn't require modifications to your C code, but if you feed it a collection of raw C header files or source code, the results might not be what you expect—in fact, they might be awful. Here's a series of steps you can follow to make an interface for a C program :

- Identify the functions that you want to wrap. It's probably not necessary to access every single function in a C program—thus, a little forethought can dramatically simplify the resulting scripting language interface. C header files are particularly good source for finding things to wrap.
- Create a new interface file to describe the scripting language interface to your program.
- Copy the appropriate declarations into the interface file or use SWIG's `%include` directive to process an entire C source/header file.
- Make sure everything in the interface file uses ANSI C/C++ syntax.

- Make sure all necessary ``typedef'` declarations and type-information is available in the interface file.
- If your program has a `main()` function, you may need to rename it (read on).
- Run SWIG and compile.

Although this may sound complicated, the process turns out to be fairly easy once you get the hang of it.

In the process of building an interface, SWIG may encounter syntax errors or other problems. The best way to deal with this is to simply copy the offending code into a separate interface file and edit it. However, the SWIG developers have worked very hard to improve the SWIG parser—you should report parsing errors to the [swig-dev mailing list](#) or to the [SWIG bug tracker](#).

### 5.7.2 The SWIG interface file

The preferred method of using SWIG is to generate separate interface file. Suppose you have the following C header file :

```
/* File : header.h */

#include <stdio.h>
#include <math.h>

extern int foo(double);
extern double bar(int, int);
extern void dump(FILE *f);
```

A typical SWIG interface file for this header file would look like the following :

```
/* File : interface.i */
%module mymodule
%{
#include "header.h"
}%
extern int foo(double);
extern double bar(int, int);
extern void dump(FILE *f);
```

Of course, in this case, our header file is pretty simple so we could have made an interface file like this as well:

```
/* File : interface.i */
%module mymodule
#include header.h
```

Naturally, your mileage may vary.

### 5.7.3 Why use separate interface files?

Although SWIG can parse many header files, it is more common to write a special `.i` file defining the interface to a package. There are several reasons why you might want to do this:

- It is rarely necessary to access every single function in a large package. Many C functions might have little or no use in a scripted environment. Therefore, why wrap them?
- Separate interface files provide an opportunity to provide more precise rules about how an interface is to be constructed.
- Interface files can provide more structure and organization.
- SWIG can't parse certain definitions that appear in header files. Having a separate file allows you to eliminate or work around these problems.
- Interface files provide a more precise definition of what the interface is. Users wanting to extend the system can go to the interface file and immediately see what is available without having to dig it out of header files.

## 5.7.4 Getting the right header files

Sometimes, it is necessary to use certain header files in order for the code generated by SWIG to compile properly. Make sure you include certain header files by using a `%{ , %}` block like this:

```
%module graphics
%{
#include <GL/gl.h>
#include <GL/glu.h>
%}

// Put rest of declarations here
...
```

## 5.7.5 What to do with main()

If your program defines a `main()` function, you may need to get rid of it or rename it in order to use a scripting language. Most scripting languages define their own `main()` procedure that is called instead. `main()` also makes no sense when working with dynamic loading. There are a few approaches to solving the `main()` conflict:

- Get rid of `main()` entirely.
- Rename `main()` to something else. You can do this by compiling your C program with an option like `-Dmain=oldmain`.
- Use conditional compilation to only include `main()` when not using a scripting language.

Getting rid of `main()` may cause potential initialization problems of a program. To handle this problem, you may consider writing a special function called `program_init()` that initializes your program upon startup. This function could then be called either from the scripting language as the first operation, or when the SWIG generated module is loaded.

As a general note, many C programs only use the `main()` function to parse command line options and to set parameters. However, by using a scripting language, you are probably trying to create a program that is more interactive. In many cases, the old `main()` program can be completely replaced by a Perl, Python, or Tcl script.

**Note:** In some cases, you might be inclined to create a scripting language wrapper for `main()`. If you do this, the compilation will probably work and your module might even load correctly. The only trouble is that when you call your `main()` wrapper, you will find that it actually invokes the `main()` of the scripting language interpreter itself! This behavior is a side effect of the symbol binding mechanism used in the dynamic linker. The bottom line: don't do this.



## 6 SWIG and C++

- [Comments on C++ Wrapping](#)
- [Approach](#)
- [Supported C++ features](#)
- [Command line options and compilation](#)
- [Simple C++ wrapping](#)
  - ◆ [Constructors and destructors](#)
  - ◆ [Default constructors](#)
  - ◆ [When constructor wrappers aren't created](#)
  - ◆ [Copy constructors](#)
  - ◆ [Member functions](#)
  - ◆ [Static members](#)
  - ◆ [Member data](#)
- [Default arguments](#)
- [Protection](#)
- [Enums and constants](#)
- [Friends](#)
- [References and pointers](#)
- [Pass and return by value](#)
- [Inheritance](#)
- [A brief discussion of multiple inheritance, pointers, and type checking](#)
- [Renaming](#)
- [Wrapping Overloaded Functions and Methods](#)
  - ◆ [Dispatch function generation](#)
  - ◆ [Ambiguity in Overloading](#)
  - ◆ [Ambiguity resolution and renaming](#)
  - ◆ [Comments on overloading](#)
- [Wrapping overloaded operators](#)
- [Class extension](#)
- [Templates](#)
- [Namespaces](#)
- [Exception specifiers](#)
- [Pointers to Members](#)
- [Smart pointers and operator->\(\)](#)
- [Using declarations and inheritance](#)
- [Partial class definitions](#)
- [A brief rant about const-correctness](#)
- [Proxy classes](#)
  - ◆ [Construction of proxy classes](#)
  - ◆ [Resource management in proxies](#)
  - ◆ [Language specific details](#)
- [Where to go for more information](#)

This chapter describes SWIG's support for wrapping C++. As a prerequisite, you should first read the chapter [SWIG Basics](#) to see how SWIG wraps ANSI C. Support for C++ builds upon ANSI C wrapping and that material will be useful in understanding this chapter.

### 6.1 Comments on C++ Wrapping

Because of its complexity and the fact that C++ can be difficult to integrate with itself let alone other languages, SWIG only provides support for a subset of C++ features. Fortunately, this is now a rather large subset.

In part, the problem with C++ wrapping is that there is no semantically obvious (or automatic ) way to map many of its advanced features into other languages. As a simple example, consider the problem of wrapping C++ multiple inheritance to a target

language with no such support. Similarly, the use of overloaded operators and overloaded functions can be problematic when no such capability exists in a target language.

A more subtle issue with C++ has to do with the way that some C++ programmers think about programming libraries. In the world of SWIG, you are really trying to create binary-level software components for use in other languages. In order for this to work, a "component" has to contain real executable instructions and there has to be some kind of binary linking mechanism for accessing its functionality. In contrast, C++ has increasingly relied upon generic programming and templates for much of its functionality. Although templates are a powerful feature, they are largely orthogonal to the whole notion of binary components and libraries. For example, an STL `vector` does not define any kind of binary object for which SWIG can just create a wrapper. To further complicate matters, these libraries often utilize a lot of behind the scenes magic in which the semantics of seemingly basic operations (e.g., pointer dereferencing, procedure call, etc.) can be changed in dramatic and sometimes non-obvious ways. Although this "magic" may present few problems in a C++-only universe, it greatly complicates the problem of crossing language boundaries and provides many opportunities to shoot yourself in the foot. You will just have to be careful.

## 6.2 Approach

To wrap C++, SWIG uses a layered approach to code generation. At the lowest level, SWIG generates a collection of procedural ANSI-C style wrappers. These wrappers take care of basic type conversion, type checking, error handling, and other low-level details of the C++ binding. These wrappers are also sufficient to bind C++ into any target language that supports built-in procedures. In some sense, you might view this layer of wrapping as providing a C library interface to C++. Optionally, SWIG can also generate proxy classes that provide a natural OO interface to the underlying code. These proxies are built on top of the low-level procedural wrappers and are typically written in the target language itself. For instance, in Python, a real Python class is used to provide a wrapper around the underlying C++ object.

It is important to emphasize that SWIG takes a deliberately conservative and non-intrusive approach to C++ wrapping. SWIG does not encapsulate C++ classes inside special C++ adaptor or proxy classes, it does not rely upon templates, nor does it use C++ inheritance when generating wrappers. The last thing that most C++ programs need is even more compiler magic. Therefore, SWIG tries to maintain a very strict and clean separation between the implementation of your C++ application and the resulting wrapper code. You might say that SWIG has been written to follow the principle of least surprise—it does not play sneaky tricks with the C++ type system, it doesn't mess with your class hierarchies, and it doesn't introduce new semantics. Although this approach might not provide the most seamless integration with C++, it is safe, simple, portable, and debuggable.

Most of this chapter focuses on the low-level procedural interface to C++ that is used as the foundation for all language modules. Keep in mind that most target languages also provide a high-level OO interface via proxy classes. A few general details about proxies can be found at the end of this chapter. However, more detailed coverage can be found in the documentation for each target language.

## 6.3 Supported C++ features

SWIG's currently supports the following C++ features :

- Classes.
- Constructors and destructors
- Virtual functions
- Public inheritance (including multiple inheritance)
- Static functions
- Function and method overloading.
- Operator overloading for many standard operators
- References
- Templates (including specialization and member templates).
- Pointers to members
- Namespaces

The following C++ features are not currently supported :

- Nested classes
- Overloaded versions of certain operators (new, delete, etc.)

SWIG's C++ support is an ongoing project so some of these limitations may be lifted in future releases. However, we make no promises. Also, submitting a bug report is a very good way to get problems fixed (wink).

## 6.4 Command line options and compilation

When wrapping C++ code, it is critical that SWIG be called with the `-c++` option. This changes the way a number of critical features such as memory management are handled. It also enables the recognition of C++ keywords. Without the `-c++` flag, SWIG will either issue a warning or a large number of syntax errors if it encounters C++ code in an interface file.

When compiling and linking the resulting wrapper file, it is normal to use the C++ compiler. For example:

```
$ swig -c++ -tcl example.i
$ c++ -c example_wrap.cxx
$ c++ example_wrap.o $(OBS) -o example.so
```

Unfortunately, the process varies slightly on each machine. Make sure you refer to the documentation on each target language for further details. The SWIG Wiki also has further details.

## 6.5 Simple C++ wrapping

The following code shows a SWIG interface file for a simple C++ class.

```
%module list
%{
#include "list.h"
%}

// Very simple C++ example for linked list

class List {
public:
    List();
    ~List();
    int  search(char *value);
    void insert(char *);
    void remove(char *);
    char *get(int n);
    int  length;
    static void print(List *l);
};
```

To generate wrappers for this class, SWIG first reduces the class to a collection of low-level C-style accessor functions. The next few sections describe this process. Later parts of the chapter describe a higher level interface based on proxy classes.

### 6.5.1 Constructors and destructors

C++ constructors and destructors are translated into accessor functions such as the following :

```
List * new_List(void) {
    return new List;
}
void delete_List(List *l) {
    delete l;
}
```

## 6.5.2 Default constructors

If a C++ class does not define any public constructors or destructors, SWIG will automatically create a default constructor or destructor. However, there are a few rules that define this behavior:

- A default constructor is not created if a class already defines a constructor with arguments.
- Default constructors are not generated for classes with pure virtual methods or for classes that inherit from an abstract class, but don't provide definitions for all of the pure methods.
- A default constructor is not created unless all bases classes support a default constructor.
- Default constructors and destructors are not created if a class defines constructors or destructors in a `private` or `protected` section.
- Default constructors and destructors are not created if any base class defines a private default constructor or a private destructor.

SWIG should never generate a constructor or destructor for a class in which it is illegal to do so. However, if it is necessary to disable the default constructor/destructor creation, the `%nodefault` directive can be used:

```
%nodefault;    // Disable creation of constructor/destructor
class Foo {
    ...
};
%makedefault;
```

`%nodefault` can also take a class name. For example:

```
%nodefault Foo;    // Disable for class Foo only.
```

**Compatibility Note:** The generation of default constructors/destructors was made the default behavior in SWIG 1.3.7. This may break certain older modules, but the old behavior can be easily restored using `%nodefault` or the `-nodefault` command line option. Furthermore, in order for SWIG to properly generate (or not generate) default constructors, it must be able to gather information from both the `private` and `protected` sections (specifically, it needs to know if a private or protected constructor/destructor is defined). In older versions of SWIG, it was fairly common to simply remove or comment out the private and protected sections of a class due to parser limitations. However, this removal may now cause SWIG to erroneously generate constructors for classes that define a constructor in those sections. Consider restoring those sections in the interface or using `%nodefault` to fix the problem.

## 6.5.3 When constructor wrappers aren't created

If a class defines a constructor, SWIG normally tries to generate a wrapper for it. However, SWIG will not generate a constructor wrapper if it thinks that it will result in illegal wrapper code. There are really two cases where this might show up.

First, SWIG won't generate wrappers for protected or private constructors. For example:

```
class Foo {
protected:
    Foo();    // Not wrapped.
public:
    ...
};
```

Next, SWIG won't generate wrappers for a class if it appears to be abstract—that is, it has undefined pure virtual methods. Here are some examples:

```
class Bar {
public:
    Bar();    // Not wrapped. Bar is abstract.
    virtual void spam(void) = 0;
};
```

```
class Grok : public Bar {
public:
    Grok();           // Not wrapped. No implementation of abstract spam().
};
```

Some users are surprised (or confused) to find missing constructor wrappers in their interfaces. In almost all cases, this is caused when classes are determined to be abstract. To see if this is the case, run SWIG with all of its warnings turned on:

```
% swig -Wall -python module.i
```

In this mode, SWIG will issue a warning for all abstract classes. It is possible to force a class to be non-abstract using this:

```
%feature("notabstract") Foo;

class Foo : public Bar {
public:
    Foo();    // Generated no matter what---not abstract.
    ...
};
```

More information about `%feature` can be found in the [Customization features](#) chapter.

## 6.5.4 Copy constructors

If a class defines more than one constructor, its behavior depends on the capabilities of the target language. If overloading is supported, the copy constructor is accessible using the normal constructor function. For example, if you have this:

```
class List {
public:
    List();
    List(const List &);    // Copy constructor
    ...
};
```

then the copy constructor can be used as follows:

```
x = new_List()           # Create a list
y = new_List(x)           # Copy list x
```

If the target language does not support overloading, then the copy constructor is available through a special function like this:

```
List *copy_List(List *f) {
    return new List(*f);
}
```

**Note:** For a class `X`, SWIG only treats a constructor as a copy constructor if it can be applied to an object of type `X` or `X *`. If more than one copy constructor is defined, only the first definition that appears is used as the copy constructor—other definitions will result in a name-clash. Constructors such as `X(const X &)`, `X(X &)`, and `X(X *)` are handled as copy constructors in SWIG.

**Note:** SWIG does *not* generate a copy constructor wrapper unless one is explicitly declared in the class. This differs from the treatment of default constructors and destructors.

**Compatibility note:** Special support for copy constructors was not added until SWIG-1.3.12. In previous versions, copy constructors could be wrapped, but they had to be renamed. For example:

```
class Foo {
public:
    Foo();
    %name(CopyFoo) Foo(const Foo &);
    ...
};
```

```
};
```

For backwards compatibility, SWIG does not perform any special copy-constructor handling if the constructor has been manually renamed. For instance, in the above example, the name of the constructor is set to `new_CopyFoo()`. This is the same as in older versions.

### 6.5.5 Member functions

All member functions are roughly translated into accessor functions like this :

```
int List_search(List *obj, char *value) {
    return obj->search(value);
}
```

This translation is the same even if the member function has been declared as `virtual`.

It should be noted that SWIG does not *actually* create a C accessor function in the code it generates. Instead, member access such as `obj->search(value)` is directly inlined into the generated wrapper functions. However, the name and calling convention of the wrappers match the accessor function prototype described above.

### 6.5.6 Static members

Static member functions are called directly without making any special transformations. For example, the static member function `print(List *l)` directly invokes `List::print(List *l)` in the generated wrapper code.

Usually, static members are accessed as functions with names in which the class name has been prepended with an underscore. For example, `List_print`.

### 6.5.7 Member data

Member data is handled in exactly the same manner as for C structures. A pair of accessor functions are created. For example :

```
int List_length_get(List *obj) {
    return obj->length;
}
int List_length_set(List *obj, int value) {
    obj->length = value;
    return value;
}
```

A read-only member can be created using the `%immutable` and `%mutable` directives. For example, we probably wouldn't want the user to change the length of a list so we could do the following to make the value available, but read-only.

```
class List {
public:
    ...
    %immutable;
    int length;
    %mutable;
    ...
};
```

Alternatively, you can specify an immutable member in advance like this:

```
%immutable List::length;
...
class List {
    ...
```

```
int length;          // Immutable by above directive
...
};
```

Similarly, all data attributes declared as `const` are wrapped as read-only members.

There are some subtle issues when wrapping data members that are themselves classes. For instance, if you had another class like this,

```
class Foo {
public:
    List items;
    ...
}
```

then access to the `items` member actually uses pointers. For example:

```
List *Foo_items_get(Foo *self) {
    return &self->items;
}
void Foo_items_set(Foo *self, List *value) {
    self->items = *value;
}
```

More information about this can be found in the "Structure data members" section of the [SWIG Basics](#) chapter.

**Compatibility note:** Read-only access used to be controlled by a pair of directives `%readonly` and `%readwrite`. Although these directives still work, they generate a warning message. Simply change the directives to `%immutable`; and `%mutable`; to silence the warning. Don't forget the extra semicolon!

**Compatibility note:** Prior to SWIG-1.3.12, all members of unknown type were wrapped into accessor functions using pointers. For example, if you had a structure like this

```
struct Foo {
    size_t len;
};
```

and nothing was known about `size_t`, then accessors would be written to work with `size_t *`. Starting in SWIG-1.3.12, this behavior has been modified. Specifically, pointers will *only* be used if SWIG knows that a datatype corresponds to a structure or class. Therefore, the above code would be wrapped into accessors involving `size_t`. This change is subtle, but it smooths over a few problems related to structure wrapping and some of SWIG's customization features.

## 6.6 Default arguments

SWIG will wrap all types of functions that have default arguments. For example member functions:

```
class Foo {
public:
    void bar(int x, int y = 3, int z = 4);
};
```

SWIG handles default arguments by generating an extra overloaded method for each defaulted argument. SWIG is effectively handling methods with default arguments as if it had wrapped the equivalent overloaded methods. Thus for the example above, it is as if we had instead given the following to SWIG:

```
class Foo {
public:
    void bar(int x, int y, int z);
    void bar(int x, int y);
    void bar(int x);
};
```

The wrappers produced are exactly the same as if the above code was instead fed into SWIG. Details of this is covered later in the [Wrapping Overloaded Functions and Methods](#) section. This approach allows SWIG to wrap all possible default arguments, but can be verbose. For example if a method has ten default arguments, then eleven wrapper methods are generated.

Please see the [Features and default arguments](#) section for more information on using `%feature` with functions with default arguments. The [Ambiguity resolution and renaming](#) section also deals with using `%rename` and `%ignore` on methods with default arguments. If you are writing your own typemaps for types used in methods with default arguments, you may also need to write a `typecheck` typemap. See the [Typemaps and overloading](#) section for details or otherwise use the `compactdefaultargs` feature as mentioned below.

**Compatibility note:** Versions of SWIG prior to SWIG-1.3.23 wrapped default arguments slightly differently. Instead a single wrapper method was generated and the default values were copied into the C++ wrappers so that the method being wrapped was then called with all the arguments specified. If the size of the wrappers are a concern then this approach to wrapping methods with default arguments can be re-activated by using the `compactdefaultargs` [feature](#).

```
%feature("compactdefaultargs") Foo::bar;
class Foo {
public:
    void bar(int x, int y = 3, int z = 4);
};
```

This is great for reducing the size of the wrappers, but the caveat is it does not work for the strongly typed languages which don't have optional arguments in the language, such as C# and Java. Another restriction of this feature is that it cannot handle default arguments that are not public. The following example illustrates this:

```
class Foo {
private:
    static const int spam;
public:
    void bar(int x, int y = spam);    // Won't work with %feature("compactdefaultargs") -
                                    // private default value
};
```

This produces uncompileable wrapper code because default values in C++ are evaluated in the same scope as the member function whereas SWIG evaluates them in the scope of a wrapper function (meaning that the values have to be public).

This feature is automatically turned on when wrapping [C code with default arguments](#) and whenever keyword arguments (kwargs) are specified for either C or C++ code. Keyword arguments are a language feature of some scripting languages, for example Ruby and Python. SWIG is unable to support kwargs when wrapping overloaded methods, so the default approach cannot be used.

## 6.7 Protection

SWIG can only wrap class members that are declared public. Anything specified in a private or protected section will simply be ignored (although the internal code generator sometimes looks at the contents of the private and protected sections so that it can properly generate code for default constructors and destructors).

By default, members of a class definition are assumed to be private until you explicitly give a `public:` declaration (This is the same convention used by C++).

## 6.8 Enums and constants

Enumerations and constants are handled differently by the different language modules and are described in detail in the appropriate language chapter. However, many languages map enums and constants in a class definition into constants with the classname as a prefix. For example :

```
class Swig {
public:
    enum {ALE, LAGER, PORTER, STOUT};
```



```
};
```

Generates the following set of constants in the target scripting language :

```
Swig_ALE = Swig::ALE
Swig_LAGER = Swig::LAGER
Swig_PORTER = Swig::PORTER
Swig_STOUT = Swig::STOUT
```

Members declared as `const` are wrapped as read-only members and do not create constants.

## 6.9 Friends

Friend declarations are ignored by SWIG. For example, if you have this code:

```
class Foo {
public:
    ...
    friend void blah(Foo *f);
    ...
};
```

then the `friend` declaration does not result in any wrapper code. On the other hand, a declaration of the function itself will work fine. For instance:

```
class Foo {
public:
    ...
    friend void blah(Foo *f);          // Ignored
    ...
};

void blah(Foo *f);                  // Generates wrappers
```

Unlike normal member functions or static member functions, a friend declaration does not define a method that operates on an instance of an object nor does it define a declaration in the scope of the class. Therefore, it would make no sense for SWIG to create wrappers as such.

## 6.10 References and pointers

C++ references are supported, but SWIG transforms them back into pointers. For example, a declaration like this :

```
class Foo {
public:
    double bar(double &a);
}
```

is accessed using a function similar to this:

```
double Foo_bar(Foo *obj, double *a) {
    obj->bar(*a);
}
```

As a special case, most language modules pass `const` references to primitive datatypes (`int`, `short`, `float`, etc.) by value instead of pointers. For example, if you have a function like this,

```
void foo(const int &x);
```

it is called from a script as follows:

```
foo(3)           # Notice pass by value
```

Functions that return a reference are remapped to return a pointer instead. For example:

```
class Bar {
public:
    Foo &spam();
};
```

Generates code like this:

```
Foo *Bar_spam(Bar *obj) {
    Foo &result = obj->spam();
    return &result;
}
```

However, functions that return `const` references to primitive datatypes (`int`, `short`, etc.) normally return the result as a value rather than a pointer. For example, a function like this,

```
const int &bar();
```

will return integers such as 37 or 42 in the target scripting language rather than a pointer to an integer.

Don't return references to objects allocated as local variables on the stack. SWIG doesn't make a copy of the objects so this will probably cause your program to crash.

**Note:** The special treatment for references to primitive datatypes is necessary to provide more seamless integration with more advanced C++ wrapping applications—especially related to templates and the STL. This was first added in SWIG-1.3.12.

## 6.11 Pass and return by value

Occasionally, a C++ program will pass and return class objects by value. For example, a function like this might appear:

```
Vector cross_product(Vector a, Vector b);
```

If no information is supplied about `Vector`, SWIG creates a wrapper function similar to the following:

```
Vector *wrap_cross_product(Vector *a, Vector *b) {
    Vector x = *a;
    Vector y = *b;
    Vector r = cross_product(x,y);
    return new Vector(r);
}
```

In order for the wrapper code to compile, `Vector` must define a copy constructor and a default constructor.

If `Vector` is defined as class in the interface, but it does not support a default constructor, SWIG changes the wrapper code by encapsulating the arguments inside a special C++ template wrapper class. This produces a wrapper that looks like this:

```
Vector cross_product(Vector *a, Vector *b) {
    SwigValueWrapper<Vector> x = *a;
    SwigValueWrapper<Vector> y = *b;
    SwigValueWrapper<Vector> r = cross_product(x,y);
    return new Vector(r);
}
```

This transformation is a little sneaky, but it provides support for pass-by-value even when a class does not provide a default constructor and it makes it possible to properly support a number of SWIG's customization options. The definition of

`SwigValueWrapper` can be found by reading the SWIG wrapper code. This class is really nothing more than a thin wrapper around a pointer.

**Note:** this transformation has no effect on typemaps or any other part of SWIG—it should be transparent except that you may see this code when reading the SWIG output file.

**Note:** This template transformation is new in SWIG-1.3.11 and may be refined in future SWIG releases. In practice, it is only necessary to do this for classes that don't define a default constructor.

**Note:** The use of this template only occurs when objects are passed or returned by value. It is not used for C++ pointers or references.

**Note:** The performance of pass-by-value is especially bad for large objects and should be avoided if possible (consider using references instead).

## 6.12 Inheritance

SWIG supports C++ public inheritance of classes and allows both single and multiple inheritance. The SWIG type-checker knows about the relationship between base and derived classes and allows pointers to any object of a derived class to be used in functions of a base class. The type-checker properly casts pointer values and is safe to use with multiple inheritance.

SWIG does not support private or protected inheritance (it is parsed, but it has no effect on the generated code). Note: private and protected inheritance do not define an "isa" relationship between classes so it would have no effect on type-checking anyways.

The following example shows how SWIG handles inheritance. For clarity, the full C++ code has been omitted.

```
// shapes.i
%module shapes
%{
#include "shapes.h"
%}

class Shape {
public:
    double x,y;
    virtual double area() = 0;
    virtual double perimeter() = 0;
    void    set_location(double x, double y);
};
class Circle : public Shape {
public:
    Circle(double radius);
    ~Circle();
    double area();
    double perimeter();
};
class Square : public Shape {
public:
    Square(double size);
    ~Square();
    double area();
    double perimeter();
}
```

When wrapped into Python, we can now perform the following operations :

```
$ python
>>> import shapes
>>> circle = shapes.new_Circle(7)
>>> square = shapes.new_Square(10)
>>> print shapes.Circle_area(circle)
```

```

153.93804004599999757
>>> print shapes.Shape_area(circle)
153.93804004599999757
>>> print shapes.Shape_area(square)
100.000000000000000000
>>> shapes.Shape_set_location(square,2,-3)
>>> print shapes.Shape_perimeter(square)
40.000000000000000000
>>>

```

In this example, Circle and Square objects have been created. Member functions can be invoked on each object by making calls to Circle\_area, Square\_area, and so on. However, the same results can be accomplished by simply using the Shape\_area function on either object.

One important point concerning inheritance is that the low-level accessor functions are only generated for classes in which they are actually declared. For instance, in the above example, the method set\_location() is only accessible as Shape\_set\_location() and not as Circle\_set\_location() or Square\_set\_location(). Of course, the Shape\_set\_location() function will accept any kind of object derived from Shape. Similarly, accessor functions for the attributes x and y are generated as Shape\_x\_get(), Shape\_x\_set(), Shape\_y\_get(), and Shape\_y\_set(). Functions such as Circle\_x\_get() are not available—instead you should use Shape\_x\_get().

Although the low-level C-like interface is functional, most language modules also produce a higher level OO interface using proxy classes. This approach is described later and can be used to provide a more natural C++ interface.

**Note:** For the best results, SWIG requires all base classes to be defined in an interface. Otherwise, you may get an warning message like this:

```
example:18. Nothing known about class 'Foo'. Ignored.
```

If any base class is undefined, SWIG still generates correct type relationships. For instance, a function accepting a Foo \* will accept any object derived from Foo regardless of whether or not SWIG actually wrapped the Foo class. If you really don't want to generate wrappers for the base class, but you want to silence the warning, you might consider using the %import directive to include the file that defines Foo. %import simply gathers type information, but doesn't generate wrappers. Alternatively, you could just define Foo as an empty class in the SWIG interface.

**Note:** typedef-names *can* be used as base classes. For example:

```

class Foo {
...
};

typedef Foo FooObj;
class Bar : public FooObj {      // Ok.  Base class is Foo
...
};

```

Similarly, typedef allows unnamed structures to be used as base classes. For example:

```

typedef struct {
...
} Foo;

class Bar : public Foo {      // Ok.
...
};

```

**Compatibility Note:** Starting in version 1.3.7, SWIG only generates low-level accessor wrappers for the declarations that are actually defined in each class. This differs from SWIG1.1 which used to inherit all of the declarations defined in base classes and regenerate specialized accessor functions such as Circle\_x\_get(), Square\_x\_get(), Circle\_set\_location(), and Square\_set\_location(). This behavior resulted in huge amounts of replicated code for large class hierarchies and made it

awkward to build applications spread across multiple modules (since accessor functions are duplicated in every single module). It is also unnecessary to have such wrappers when advanced features like proxy classes are used. Future versions of SWIG may apply further optimizations such as not regenerating wrapper functions for virtual members that are already defined in a base class.

## 6.13 A brief discussion of multiple inheritance, pointers, and type checking

When a target scripting language refers to a C++ object, it normally uses a tagged pointer object that contains both the value of the pointer and a type string. For example, in Tcl, a C++ pointer might be encoded as a string like this:

```
_808fea88_p_Circle
```

A somewhat common question is whether or not the type-tag could be safely removed from the pointer. For instance, to get better performance, could you strip all type tags and just use simple integers instead?

In general, the answer to this question is no. In the wrappers, all pointers are converted into a common data representation in the target language. Typically this is the equivalent of casting a pointer to `void *`. This means that any C++ type information associated with the pointer is lost in the conversion.

The problem with losing type information is that it is needed to properly support many advanced C++ features—especially multiple inheritance. For example, suppose you had code like this:

```
class A {
public:
    int x;
};

class B {
public:
    int y;
};

class C : public A, public B {
};

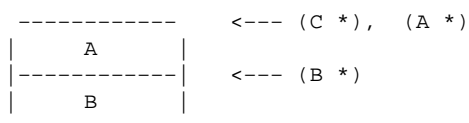
int A_function(A *a) {
    return a->x;
}

int B_function(B *b) {
    return b->y;
}
```

Now, consider the following code that uses `void *`.

```
C *c = new C();
void *p = (void *) c;
...
int x = A_function((A *) p);
int y = B_function((B *) p);
```

In this code, both `A_function()` and `B_function()` may legally accept an object of type `C *` (via inheritance). However, one of the functions will always return the wrong result when used as shown. The reason for this is that even though `p` points to an object of type `C`, the casting operation doesn't work like you would expect. Internally, this has to do with the data representation of `C`. With multiple inheritance, the data from each base class is stacked together. For example:



Because of this stacking, a pointer of type `C *` may change value when it is converted to a `A *` or `B *`. However, this adjustment does *not* occur if you are converting from a `void *`.

The use of type tags marks all pointers with the real type of the underlying object. This extra information is then used by SWIG generated wrappers to correctly cast pointer values under inheritance (avoiding the above problem).

One might be inclined to fix this problem using some variation of `dynamic_cast<>`. The only problem is that it doesn't work with `void` pointers, it requires RTTI support, and it only works with polymorphic classes (i.e., classes that define one or more virtual functions).

The bottom line: learn to live with type-tagged pointers.

## 6.14 Renaming

C++ member functions and data can be renamed with the `%name` directive. The `%name` directive only replaces the member function name. For example :

```
class List {
public:
    List();
    %name(ListSize) List(int maxsize);
    ~List();
    int search(char *value);
    %name(find) void insert(char *);
    %name(delete) void remove(char *);
    char *get(int n);
    int length;
    static void print(List *l);
};
```

This will create the functions `List_find`, `List_delete`, and a function named `new_ListSize` for the overloaded constructor.

The `%name` directive can be applied to all members including constructors, destructors, static functions, data members, and enumeration values.

The class name prefix can also be changed by specifying

```
%name(newname) class List {
...
}
```

Although the `%name( )` directive can be used to help deal with overloaded methods, it really doesn't work very well because it requires a lot of additional markup in your interface. Keep reading for a better solution.

## 6.15 Wrapping Overloaded Functions and Methods

In many language modules, SWIG provides partial support for overloaded functions, methods, and constructors. For example, if you supply SWIG with overloaded functions like this:

```
void foo(int x) {
    printf("x is %d\n", x);
}
void foo(char *x) {
    printf("x is '%s'\n", x);
}
```

The function is used in a completely natural way. For example:

```
>>> foo(3)
x is 3
>>> foo("hello")
x is 'hello'
>>>
```

Overloading works in a similar manner for methods and constructors. For example if you have this code,

```
class Foo {
public:
    Foo();
    Foo(const Foo &);    // Copy constructor
    void bar(int x);
    void bar(char *s, int y);
};
```

it might be used like this

```
>>> f = Foo()           # Create a Foo
>>> f.bar(3)
>>> g = Foo(f)          # Copy Foo
>>> f.bar("hello",2)
```

### 6.15.1 Dispatch function generation

The implementation of overloaded functions and methods is somewhat complicated due to the dynamic nature of scripting languages. Unlike C++, which binds overloaded methods at compile time, SWIG must determine the proper function as a runtime check for scripting language targets. This check is further complicated by the typeless nature of certain scripting languages. For instance, in Tcl, all types are simply strings. Therefore, if you have two overloaded functions like this,

```
void foo(char *x);
void foo(int x);
```

the order in which the arguments are checked plays a rather critical role.

For statically typed languages, SWIG uses the language's method overloading mechanism. To implement overloading for the scripting languages, SWIG generates a dispatch function that checks the number of passed arguments and their types. To create this function, SWIG first examines all of the overloaded methods and ranks them according to the following rules:

1. **Number of required arguments.** Methods are sorted by increasing number of required arguments.
2. **Argument type precedence.** All C++ datatypes are assigned a numeric type precedence value (which is determined by the language module).

Type	Precedence
-----	-----
TYPE *	0 (High)
void *	20
Integers	40
Floating point	60
char	80
Strings	100 (Low)

Using these precedence values, overloaded methods with the same number of required arguments are sorted in increased order of precedence values.

This may sound very confusing, but an example will help. Consider the following collection of overloaded methods:

```
void foo(double);
void foo(int);
void foo(Bar *);
void foo();
```

```
void foo(int x, int y, int z, int w);
void foo(int x, int y, int z = 3);
void foo(double x, double y);
void foo(double x, Bar *z);
```

The first rule simply ranks the functions by required argument count. This would produce the following list:

```
rank
-----
[0]   foo()
[1]   foo(double);
[2]   foo(int);
[3]   foo(Bar *);
[4]   foo(int x, int y, int z = 3);
[5]   foo(double x, double y)
[6]   foo(double x, Bar *z)
[7]   foo(int x, int y, int z, int w);
```

The second rule, simply refines the ranking by looking at argument type precedence values.

```
rank
-----
[0]   foo()
[1]   foo(Bar *);
[2]   foo(int);
[3]   foo(double);
[4]   foo(int x, int y, int z = 3);
[5]   foo(double x, Bar *z)
[6]   foo(double x, double y)
[7]   foo(int x, int y, int z, int w);
```

Finally, to generate the dispatch function, the arguments passed to an overloaded method are simply checked in the same order as they appear in this ranking.

If you're still confused, don't worry about it—SWIG is probably doing the right thing.

## 6.15.2 Ambiguity in Overloading

Regrettably, SWIG is not able to support every possible use of valid C++ overloading. Consider the following example:

```
void foo(int x);
void foo(long x);
```

In C++, this is perfectly legal. However, in a scripting language, there is generally only one kind of integer object. Therefore, which one of these functions do you pick? Clearly, there is no way to truly make a distinction just by looking at the value of the integer itself (`int` and `long` may even be the same precision). Therefore, when SWIG encounters this situation, it may generate a warning message like this for scripting languages:

```
example.i:4: Warning(509): Overloaded foo(long) is shadowed by foo(int) at example.i:3.
```

or for statically typed languages like Java:

```
example.i:4: Warning(516): Overloaded method foo(long) ignored. Method foo(int)
at example.i:3 used.
```

This means that the second overloaded function will be inaccessible from a scripting interface or the method won't be wrapped at all. This is done as SWIG does not know how to disambiguate it from an earlier method.

Ambiguity problems are known to arise in the following situations:



## SWIG-1.3 Documentation

- Integer conversions. Datatypes such as `int`, `long`, and `short` cannot be disambiguated in some languages. Shown above.
- Floating point conversion. `float` and `double` can not be disambiguated in some languages.
- Pointers and references. For example, `Foo *` and `Foo &`.
- Pointers and arrays. For example, `Foo *` and `Foo [4]`.
- Pointers and instances. For example, `Foo` and `Foo *`. Note: SWIG converts all instances to pointers.
- Qualifiers. For example, `const Foo *` and `Foo *`.
- Default vs. non default arguments. For example, `foo(int a, int b)` and `foo(int a, int b = 3)`.

When an ambiguity arises, methods are checked in the same order as they appear in the interface file. Therefore, earlier methods will shadow methods that appear later.

When wrapping an overloaded function, there is a chance that you will get an error message like this:

```
example.i:3: Warning(467): Overloaded foo(int) not supported (no type checking
rule for 'int').
```

This error means that the target language module supports overloading, but for some reason there is no type-checking rule that can be used to generate a working dispatch function. The resulting behavior is then undefined. You should report this as a bug to the [SWIG bug tracking database](#).

If you get an error message such as the following,

```
foo.i:6. Overloaded declaration ignored.  Spam::foo(double )
foo.i:5. Previous declaration is Spam::foo(int )
foo.i:7. Overloaded declaration ignored.  Spam::foo(Bar *,Spam *,int )
foo.i:5. Previous declaration is Spam::foo(int )
```

it means that the target language module has not yet implemented support for overloaded functions and methods. The only way to fix the problem is to read the next section.

### 6.15.3 Ambiguity resolution and renaming

If an ambiguity in overload resolution occurs or if a module doesn't allow overloading, there are a few strategies for dealing with the problem. First, you can tell SWIG to ignore one of the methods. This is easy—simply use the `%ignore` directive. For example:

```
%ignore foo(long);

void foo(int);
void foo(long);          // Ignored.  Oh well.
```

The other alternative is to rename one of the methods. This can be done using `%rename`. For example:

```
%rename(foo_long) foo(long);

void foo(int);
void foo(long);          // Accessed as foo_long()
```

The `%ignore` and `%rename` directives are both rather powerful in their ability to match declarations. When used in their simple form, they apply to both global functions and methods. For example:

```
/* Forward renaming declarations */
%rename(foo_i) foo(int);
%rename(foo_d) foo(double);
...
void foo(int);          // Becomes 'foo_i'
void foo(char *c);      // Stays 'foo' (not renamed)

class Spam {
```

```
public:
    void foo(int);        // Becomes 'foo_i'
    void foo(double);     // Becomes 'foo_d'
    ...
};
```

If you only want the renaming to apply to a certain scope, the C++ scope resolution operator (::) can be used. For example:

```
%rename(foo_i) ::foo(int);        // Only rename foo(int) in the global scope.
                                   // (will not rename class members)

%rename(foo_i) Spam::foo(int);    // Only rename foo(int) in class Spam
```

When a renaming operator is applied to a class as in `Spam::foo(int)`, it is applied to that class and all derived classes. This can be used to apply a consistent renaming across an entire class hierarchy with only a few declarations. For example:

```
%rename(foo_i) Spam::foo(int);
%rename(foo_d) Spam::foo(double);

class Spam {
public:
    virtual void foo(int);        // Renamed to foo_i
    virtual void foo(double);     // Renamed to foo_d
    ...
};

class Bar : public Spam {
public:
    virtual void foo(int);        // Renamed to foo_i
    virtual void foo(double);     // Renamed to foo_d
    ...
};

class Grok : public Bar {
public:
    virtual void foo(int);        // Renamed to foo_i
    virtual void foo(double);     // Renamed to foo_d
    ...
};
```

It is also possible to include `%rename` specifications in the class definition itself. For example:

```
class Spam {
    %rename(foo_i) foo(int);
    %rename(foo_d) foo(double);
public:
    virtual void foo(int);        // Renamed to foo_i
    virtual void foo(double);     // Renamed to foo_d
    ...
};

class Bar : public Spam {
public:
    virtual void foo(int);        // Renamed to foo_i
    virtual void foo(double);     // Renamed to foo_d
    ...
};
```

In this case, the `%rename` directives still get applied across the entire inheritance hierarchy, but it's no longer necessary to explicitly specify the class prefix `Spam::`.

A special form of `%rename` can be used to apply a renaming just to class members (of all classes):

```
%rename(foo_i) *::foo(int);    // Only rename foo(int) if it appears in a class.
```

Note: the `*::` syntax is non-standard C++, but the `'*` is meant to be a wildcard that matches any class name (we couldn't think of a better alternative so if you have a better idea, send email to the [swig-dev mailing list](#)).

Although this discussion has primarily focused on `%rename` all of the same rules also apply to `%ignore`. For example:

```
%ignore foo(double);           // Ignore all foo(double)
%ignore Spam::foo;             // Ignore foo in class Spam
%ignore Spam::foo(double);     // Ignore foo(double) in class Spam
%ignore *::foo(double);       // Ignore foo(double) in all classes
```

When applied to a base class, `%ignore` forces all definitions in derived classes to disappear. For example, `%ignore Spam::foo(double)` will eliminate `foo(double)` in `Spam` and all classes derived from `Spam`.

#### Notes on `%rename` and `%ignore`:

- Since, the `%rename` declaration is used to declare a renaming in advance, it can be placed at the start of an interface file. This makes it possible to apply a consistent name resolution without having to modify header files. For example:

```
%module foo

/* Rename these overloaded functions */
%rename(foo_i) foo(int);
%rename(foo_d) foo(double);

#include "header.h"
```

- The scope qualifier (`::`) can also be used on simple names. For example:

```
%rename(bar) ::foo;           // Rename foo to bar in global scope only
%rename(bar) Spam::foo;       // Rename foo to bar in class Spam only
%rename(bar) *::foo;          // Rename foo in classes only
```

- Name matching tries to find the most specific match that is defined. A qualified name such as `Spam::foo` always has higher precedence than an unqualified name `foo`. `Spam::foo` has higher precedence than `*::foo` and `*::foo` has higher precedence than `foo`. A parameterized name has higher precedence than an unparameterized name within the same scope level. However, an unparameterized name with a scope qualifier has higher precedence than a parameterized name in global scope (e.g., a renaming of `Spam::foo` takes precedence over a renaming of `foo(int)`).
- The order in which `%rename` directives are defined does not matter as long as they appear before the declarations to be renamed. Thus, there is no difference between saying:

```
%rename(bar) foo;
%rename(foo_i) Spam::foo(int);
%rename(Foo) Spam::foo;
```

and this

```
%rename(Foo) Spam::foo;
%rename(bar) foo;
%rename(foo_i) Spam::foo(int);
```

(the declarations are not stored in a linked list and order has no importance). Of course, a repeated `%rename` directive will change the setting for a previous `%rename` directive if exactly the same name, scope, and parameters are supplied.

- For multiple inheritance where renaming rules are defined for multiple base classes, the first renaming rule found on a depth-first traversal of the class hierarchy is used.
- The name matching rules strictly follow member qualification rules. For example, if you have a class like this:

```
class Spam {
public:
    ...
    void bar() const;
    ...
};
```

the declaration

```
%rename(name) Spam::bar();
```

will not apply as there is no unqualified member `bar()`. The following will apply as the qualifier matches correctly:

```
%rename(name) Spam::bar() const;
```

An often overlooked C++ feature is that classes can define two different overloaded members that differ only in their qualifiers, like this:

```
class Spam {
public:
    ...
    void bar();           // Unqualified member
    void bar() const;     // Qualified member
    ...
};
```

`%rename` can then be used to target each of the overloaded methods individually. For example we can give them separate names in the target language:

```
%rename(name1) Spam::bar();
%rename(name2) Spam::bar() const;
```

Similarly, if you merely wanted to ignore one of the declarations, use `%ignore` with the full qualification. For example, the following directive would tell SWIG to ignore the `const` version of `bar()` above:

```
%ignore Spam::bar() const; // Ignore bar() const, but leave other bar() alone
```

- The name matching rules also use default arguments for finer control when wrapping methods that have default arguments. Recall that methods with default arguments are wrapped as if the equivalent overloaded methods had been parsed ([Default arguments](#) section). Let's consider the following example class:

```
class Spam {
public:
    ...
    void bar(int i=-1, double d=0.0);
    ...
};
```

The following `%rename` will match exactly and apply to all the target language overloaded methods because the declaration with the default arguments exactly matches the wrapped method:

```
%rename(newbar) Spam::bar(int i=-1, double d=0.0);
```

The C++ method can then be called from the target language with the new name no matter how many arguments are specified, for example: `newbar(2, 2.0)`, `newbar(2)` or `newbar()`. However, if the `%rename` does not contain the default arguments, it will only apply to the single equivalent target language overloaded method. So if instead we have:

```
%rename(newbar) Spam::bar(int i, double d);
```

The C++ method must then be called from the target language with the new name `newbar(2, 2.0)` when both arguments are supplied or with the original name as `bar(2)` (one argument) or `bar()` (no arguments). In fact it is possible to use `%rename` on the equivalent overloaded methods, to rename all the equivalent overloaded methods:

```
%rename(bar_2args) Spam::bar(int i, double d);
%rename(bar_larg) Spam::bar(int i);
%rename(bar_default) Spam::bar();
```

Similarly, the extra overloaded methods can be selectively ignored using `%ignore`.

**Compatibility note:** The `%rename` directive introduced the default argument matching rules in SWIG-1.3.23 at the same time as the changes to wrapping methods with default arguments was introduced.

## 6.15.4 Comments on overloading

Support for overloaded methods was first added in SWIG-1.3.14. The implementation is somewhat unusual when compared to similar tools. For instance, the order in which declarations appear is largely irrelevant in SWIG. Furthermore, SWIG does not rely upon trial execution or exception handling to figure out which method to invoke.

Internally, the overloading mechanism is completely configurable by the target language module. Therefore, the degree of overloading support may vary from language to language. As a general rule, statically typed languages like Java are able to provide more support than dynamically typed languages like Perl, Python, Ruby, and Tcl.

## 6.16 Wrapping overloaded operators

Starting in SWIG-1.3.10, C++ overloaded operator declarations can be wrapped. For example, consider a class like this:

```
class Complex {
private:
    double rpart, ipart;
public:
    Complex(double r = 0, double i = 0) : rpart(r), ipart(i) { }
    Complex(const Complex &c) : rpart(c.rpart), ipart(c.ipart) { }
    Complex &operator=(const Complex &c) {
        rpart = c.rpart;
        ipart = c.ipart;
        return *this;
    }
    Complex operator+(const Complex &c) const {
        return Complex(rpart+c.rpart, ipart+c.ipart);
    }
    Complex operator-(const Complex &c) const {
        return Complex(rpart-c.rpart, ipart-c.ipart);
    }
    Complex operator*(const Complex &c) const {
        return Complex(rpart*c.rpart - ipart*c.ipart,
                       rpart*c.ipart + c.rpart*ipart);
    }
    Complex operator-() const {
        return Complex(-rpart, -ipart);
    }
    double re() const { return rpart; }
    double im() const { return ipart; }
};
```

When operator declarations appear, they are handled in *exactly* the same manner as regular methods. However, the names of these methods are set to strings like "operator +" or "operator -". The problem with these names is that they are illegal identifiers in most scripting languages. For instance, you can't just create a method called "operator +" in Python—there won't be any way to call it.

Some language modules already know how to automatically handle certain operators (mapping them into operators in the target language). However, the underlying implementation of this is really managed in a very general way using the `%rename` directive. For example, in Python a declaration similar to this is used:

```
%rename(__add__) Complex::operator+;
```

This binds the + operator to a method called `__add__` (which is conveniently the same name used to implement the Python + operator). Internally, the generated wrapper code for a wrapped operator will look something like this pseudocode:

```

_wrap_Complex__add__(args) {
    ... get args ...
    obj->operator+(args);
    ...
}

```

When used in the target language, it may now be possible to use the overloaded operator normally. For example:

```

>>> a = Complex(3,4)
>>> b = Complex(5,2)
>>> c = a + b           # Invokes __add__ method

```

It is important to realize that there is nothing magical happening here. The `%rename` directive really only picks a valid method name. If you wrote this:

```
%rename(add) operator+;
```

The resulting scripting interface might work like this:

```

a = Complex(3,4)
b = Complex(5,2)
c = a.add(b)      # Call a.operator+(b)

```

All of the techniques described to deal with overloaded functions also apply to operators. For example:

```

%ignore Complex::operator=;           // Ignore = in class Complex
%ignore *::operator=;                 // Ignore = in all classes
%ignore operator=;                    // Ignore = everywhere.

%rename(__sub__) Complex::operator-;
%rename(__neg__) Complex::operator-(); // Unary -

```

The last part of this example illustrates how multiple definitions of the `operator-` method might be handled.

Handling operators in this manner is mostly straightforward. However, there are a few subtle issues to keep in mind:

- In C++, it is fairly common to define different versions of the operators to account for different types. For example, a class might also include a friend function like this:

```

class Complex {
public:
    friend Complex operator+(Complex &, double);
};
Complex operator+(Complex &, double);

```

SWIG simply ignores all `friend` declarations. Furthermore, it doesn't know how to associate the associated `operator+` with the class (because it's not a member of the class).

It's still possible to make a wrapper for this operator, but you'll have to handle it like a normal function. For example:

```
%rename(add_complex_double) operator+(Complex &, double);
```

- Certain operators are ignored by default. For instance, `new` and `delete` operators are ignored as well as conversion operators.
- The semantics of certain C++ operators may not match those in the target language.

## 6.17 Class extension

New methods can be added to a class using the `%extend` directive. This directive is primarily used in conjunction with proxy classes to add additional functionality to an existing class. For example :

```

%module vector
%{
#include "vector.h"
%}

class Vector {
public:
    double x,y,z;
    Vector();
    ~Vector();
    ... bunch of C++ methods ...
    %extend {
        char *__str__() {
            static char temp[256];
            sprintf(temp,"[ %g, %g, %g ]", v->x,v->y,v->z);
            return &temp[0];
        }
    }
};

```

This code adds a `__str__` method to our class for producing a string representation of the object. In Python, such a method would allow us to print the value of an object using the `print` command.

```

>>>
>>> v = Vector();
>>> v.x = 3
>>> v.y = 4
>>> v.z = 0
>>> print(v)
[ 3.0, 4.0, 0.0 ]
>>>

```

The `%extend` directive follows all of the same conventions as its use with C structures. Please refer to the [SWIG Basics](#) chapter for further details.

**Compatibility note:** The `%extend` directive is a new name for the `%addmethods` directive. Since `%addmethods` could be used to extend a structure with more than just methods, a more suitable directive name has been chosen.

## 6.18 Templates

In all versions of SWIG, template type names may appear anywhere a type is expected in an interface file. For example:

```

void foo(vector<int> *a, int n);
void bar(list<int,100> *x);

```

There are some restrictions on the use of non-type arguments. Specifically, they have to be simple literals and not expressions. For example:

```

void bar(list<int,100> *x);    // OK
void bar(list<int,2*50> *x);  // Illegal

```

The type system is smart enough to figure out clever games you might try to play with `typedef`. For instance, consider this code:

```

typedef int Integer;
void foo(vector<int> *x, vector<Integer> *y);

```

In this case, `vector<Integer>` is exactly the same type as `vector<int>`. The wrapper for `foo()` will accept either variant.

Starting with SWIG-1.3.7, simple C++ template declarations can also be wrapped. SWIG-1.3.12 greatly expands upon the earlier implementation. Before discussing this any further, there are a few things you need to know about template wrapping. First, a bare C++ template does not define any sort of runnable object-code for which SWIG can normally create a wrapper. Therefore, in order to wrap a template, you need to give SWIG information about a particular template instantiation (e.g., `vector<int>`, `array<double>`, etc.). Second, an instantiation name such as `vector<int>` is generally not a valid identifier name in most target languages. Thus, you will need to give the template instantiation a more suitable name such as `intvector` when creating a wrapper.

To illustrate, consider the following template definition:

```
template<class T> class List {
private:
    T *data;
    int nitems;
    int maxitems;
public:
    List(int max) {
        data = new T [max];
        nitems = 0;
        maxitems = max;
    }
    ~List() {
        delete [] data;
    };
    void append(T obj) {
        if (nitems < maxitems) {
            data[nitems++] = obj;
        }
    }
    int length() {
        return nitems;
    }
    T get(int n) {
        return data[n];
    }
};
```

By itself, this template declaration is useless—SWIG simply ignores it because it doesn't know how to generate any code until unless a definition of `T` is provided.

One way to create wrappers for a specific template instantiation is to simply provide an expanded version of the class directly like this:

```
%rename(intList) List<int>;          // Rename to a suitable identifier
class List<int> {
private:
    int *data;
    int nitems;
    int maxitems;
public:
    List(int max);
    ~List();
    void append(int obj);
    int length();
    int get(int n);
};
```

The `%rename` directive is needed to give the template class an appropriate identifier name in the target language (most languages would not recognize C++ template syntax as a valid class name). The rest of the code is the same as what would appear in a normal class definition.

Since manual expansion of templates gets old in a hurry, the `%template` directive can be used to create instantiations of a template class. Semantically, `%template` is simply a shortcut—it expands template code in exactly the same way as shown



above. Here are some examples:

```
/* Instantiate a few different versions of the template */
%template(intList) List<int>;
%template(doubleList) List<double>;
```

The argument to `%template()` is the name of the instantiation in the target language. The name you choose should not conflict with any other declarations in the interface file with one exception—it is okay for the template name to match that of a typedef declaration. For example:

```
%template(intList) List<int>;
...
typedef List<int> intList;    // OK
```

SWIG can also generate wrappers for function templates using a similar technique. For example:

```
// Function template
template<class T> T max(T a, T b) { return a > b ? a : b; }

// Make some different versions of this function
%template(maxint) max<int>;
%template(maxdouble) max<double>;
```

In this case, `maxint` and `maxdouble` become unique names for specific instantiations of the function.

The number of arguments supplied to `%template` should match that in the original template definition. Template default arguments are supported. For example:

```
template vector<typename T, int max=100> class vector {
...
};

%template(intvec) vector<int>;           // OK
%template(vec1000) vector<int,1000>;    // OK
```

The `%template` directive should not be used to wrap the same template instantiation more than once in the same scope. This will generate an error. For example:

```
%template(intList) List<int>;
%template(Listint) List<int>;    // Error.    Template already wrapped.
```

This error is caused because the template expansion results in two identical classes with the same name. This generates a symbol table conflict. Besides, it is probably more efficient to only wrap a specific instantiation only once in order to reduce the potential for code bloat.

Since the type system knows how to handle typedef, it is generally not necessary to instantiate different versions of a template for typenames that are equivalent. For instance, consider this code:

```
%template(intList) vector<int>;
typedef int Integer;
...
void foo(vector<Integer> *x);
```

In this case, `vector<Integer>` is exactly the same type as `vector<int>`. Any use of `Vector<Integer>` is mapped back to the instantiation of `vector<int>` created earlier. Therefore, it is not necessary to instantiate a new class for the type `Integer` (doing so is redundant and will simply result in code bloat).

When a template is instantiated using `%template`, information about that class is saved by SWIG and used elsewhere in the program. For example, if you wrote code like this,

```
...
%template(intList) List<int>;
...
class UltraList : public List<int> {
    ...
};
```

then SWIG knows that `List<int>` was already wrapped as a class called `intList` and arranges to handle the inheritance correctly. If, on the other hand, nothing is known about `List<int>`, you will get a warning message similar to this:

```
example.h:42. Nothing known about class 'List<int >' (ignored).
example.h:42. Maybe you forgot to instantiate 'List<int >' using %template.
```

If a template class inherits from another template class, you need to make sure that base classes are instantiated before derived classes. For example:

```
template<class T> class Foo {
    ...
};

template<class T> class Bar : public Foo<T> {
    ...
};

// Instantiate base classes first
%template(intFoo) Foo<int>;
%template(doubleFoo) Foo<double>;

// Now instantiate derived classes
%template(intBar) Bar<int>;
%template(doubleBar) Bar<double>;
```

The order is important since SWIG uses the instantiation names to properly set up the inheritance hierarchy in the resulting wrapper code (and base classes need to be wrapped before derived classes). Don't worry—if you get the order wrong, SWIG should generate a warning message.

Occasionally, you may need to tell SWIG about base classes that are defined by templates, but which aren't supposed to be wrapped. Since SWIG is not able to automatically instantiate templates for this purpose, you must do it manually. To do this, simply use `%template` with no name. For example:

```
// Instantiate traits<double,double>, but don't wrap it.
%template() traits<double,double>;
```

If you have to instantiate a lot of different classes for many different types, you might consider writing a SWIG macro. For example:

```
%define TEMPLATE_WRAP(T,prefix)
%template(prefix ## Foo) Foo<T>;
%template(prefix ## Bar) Bar<T>;
...
%enddef

TEMPLATE_WRAP(int, int)
TEMPLATE_WRAP(double, double)
TEMPLATE_WRAP(char *, String)
...
```

The SWIG template mechanism *does* support specialization. For instance, if you define a class like this,

```
template<> class List<int> {
private:
    int *data;
    int nitems;
```

```

    int maxitems;
public:
    List(int max);
    ~List();
    void append(int obj);
    int length();
    int get(int n);
};

```

then SWIG will use this code whenever the user expands `List<int>`. In practice, this may have very little effect on the underlying wrapper code since specialization is often used to provide slightly modified method bodies (which are ignored by SWIG). However, special SWIG directives such as `%typemap`, `%extend`, and so forth can be attached to a specialization to provide customization for specific types.

Partial template specialization is partially supported by SWIG. For example, this code defines a template that is applied when the template argument is a pointer.

```

template<class T> class List<T*> {
private:
    T *data;
    int nitems;
    int maxitems;
public:
    List(int max);
    ~List();
    void append(int obj);
    int length();
    T get(int n);
};

```

SWIG should be able to handle most simple uses of partial specialization. However, it may fail to match templates properly in more complicated cases. For example, if you have this code,

```

template<class T1, class T2> class Foo<T1, T2 *> { };

```

SWIG isn't able to match it properly for instantiations like `Foo<int *, int *>`. This problem is not due to parsing, but due to the fact that SWIG does not currently implement all of the C++ argument deduction rules.

Member function templates are supported. The underlying principle is the same as for normal templates—SWIG can't create a wrapper unless you provide more information about types. For example, a class with a member template might look like this:

```

class Foo {
public:
    template<class T> void bar(T x, T y) { ... };
    ...
};

```

To expand the template, simply use `%template` inside the class.

```

class Foo {
public:
    template<class T> void bar(T x, T y) { ... };
    ...
    %template(barint)    bar<int>;
    %template(bardouble) bar<double>;
};

```

Or, if you want to leave the original class definition alone, just do this:

```

class Foo {
public:
    template<class T> void bar(T x, T y) { ... };
};

```

```

    ...
};
...
%extend Foo {
    %template(barint)    bar<int>;
    %template(bardouble) bar<double>;
};

```

Note: because of the way that templates are handled, the `%template` directive must always appear *after* the definition of the template to be expanded.

When used with members, the `%template` directive may be placed in another template class. Here is a slightly perverse example:

```

// A template
template<class T> class Foo {
public:
    // A member template
    template<class S> T bar(S x, S y) { ... };
    ...
};

// Expand a few member templates
%extend Foo {
    %template(bari) bar<int>;
    %template(bard) bar<double>;
};

// Create some wrappers for the template
%template(Fooi) Foo<int>;
%template(Food) Foo<double>;

```

Miraculously, you will find that each expansion of `Foo` has member functions `bari()` and `bard()` added.

A common use of member templates is to define constructors for copies and conversions. For example:

```

template<class T1, class T2> struct pair {
    T1 first;
    T2 second;
    pair() : first(T1()), second(T2()) { }
    pair(const T1 &x, const T2 &y) : first(x), second(y) { }
    template<class U1, class U2> pair(const pair<U1,U2> &x)
                                   : first(x.first),second(x.second) { }
};

```

This declaration is perfectly acceptable to SWIG, but the constructor template will be ignored unless you explicitly expand it. To do that, you could expand a few versions of the constructor in the template class itself. For example:

```

%extend pair {
    %template(pair) pair<T1,T2>;          // Generate default copy constructor
};

```

When using `%extend` in this manner, notice how you can still use the template parameters in the original template definition.

Alternatively, you could expand the constructor template in selected instantiations. For example:

```

// Instantiate a few versions
%template(pairii) pair<int,int>;
%template(pairdd) pair<double,double>;

// Create a conversion constructor from int to double
%extend pair<double,double> {
    %template(pairdd_from_pairii) pair<int,int>;    // Conversion constructor
}

```

```
};
```

Admittedly, this isn't very pretty or automatic. However, it's probably better than nothing—well, maybe.

If all of this isn't quite enough and you really want to make someone's head explode, SWIG directives such as `%rename`, `%extend`, and `%typemap` can be included directly in template definitions. For example:

```
// File : list.h
template<class T> class List {
...
public:
    %rename(__getitem__) get(int);
    List(int max);
    ~List();
    ...
    T get(int index);
    %extend {
        char *__str__() {
            /* Make a string representation */
            ...
        }
    }
};
```

In this example, the extra SWIG directives are propagated to *every* template instantiation.

It is also possible to separate these declarations from the template class. For example:

```
%rename(__getitem__) List::get;
%extend List {
    char *__str__() {
        /* Make a string representation */
        ...
    }
    /* Make a copy */
    T *__copy__() {
        return new List<T>(*self);
    }
};

...
template<class T> class List {
...
public:
    List() { };
    ...
};
```

When `%extend` is decoupled from the class definition, it is legal to use the same template parameters as provided in the class definition. These are replaced when the template is expanded. In addition, the `%extend` directive can be used to add additional methods to a specific instantiation. For example:

```
%template(intList) List<int>;

%extend List<int> {
    void blah() {
        printf("Hey, I'm an List<int>!\n");
    }
};
```

SWIG even supports overloaded templated functions. As usual the `%template` directive is used to wrap templated functions. For example:

```
template<class T> void foo(T x) { };
```

```
template<class T> void foo(T x, T y) { };

%template(foo) foo<int>;
```

This will generate two overloaded wrapper methods, the first will take a single integer as an argument and the second will take two integer arguments.

Needless to say, SWIG's template support provides plenty of opportunities to break the universe. That said, an important final point is that **SWIG does not perform extensive error checking of templates!** Specifically, SWIG does not perform type checking nor does it check to see if the actual contents of the template declaration make any sense. Since the C++ compiler will hopefully check this when it compiles the resulting wrapper file, there is no practical reason for SWIG to duplicate this functionality (besides, none of the SWIG developers are masochistic enough to want to implement this right now).

**Compatibility Note:** The first implementation of template support relied heavily on macro expansion in the preprocessor. Templates have been more tightly integrated into the parser and type system in SWIG-1.3.12 and the preprocessor is no longer used. Code that relied on preprocessing features in template expansion will no longer work. However, SWIG still allows the # operator to be used to generate a string from a template argument.

**Compatibility Note:** In earlier versions of SWIG, the %template directive introduced a new class name. This name could then be used with other directives. For example:

```
%template(vectori) vector<int>;
%extend vectori {
    void somemethod() { }
};
```

This behavior is no longer supported. Instead, you should use the original template name as the class name. For example:

```
%template(vectori) vector<int>;
%extend vector<int> {
    void somemethod() { }
};
```

Similar changes apply to typemaps and other customization features.

## 6.19 Namespaces

Support for C++ namespaces is a relatively late addition to SWIG, first appearing in SWIG-1.3.12. Before describing the implementation, it is worth noting that the semantics of C++ namespaces is extremely non-trivial—especially with regard to the C++ type system and class machinery. At a most basic level, namespaces are sometimes used to encapsulate common functionality. For example:

```
namespace math {
    double sin(double);
    double cos(double);

    class Complex {
        double im, re;
    public:
        ...
    };
    ...
};
```

Members of the namespace are accessed in C++ by prepending the namespace prefix to names. For example:

```
double x = math::sin(1.0);
double magitude(math::Complex *c);
math::Complex c;
...
```

At this level, namespaces are relatively easy to manage. However, things start to get very ugly when you throw in the other ways a namespace can be used. For example, selective symbols can be exported from a namespace with `using`.

```
using math::Complex;
double magnitude(Complex *c);          // Namespace prefix stripped
```

Similarly, the contents of an entire namespace can be made available like this:

```
using namespace math;
double x = sin(1.0);
double magnitude(Complex *c);
```

Alternatively, a namespace can be aliased:

```
namespace M = math;
double x = M::sin(1.0);
double magnitude(M::Complex *c);
```

Using combinations of these features, it is possible to write head-exploding code like this:

```
namespace A {
    class Foo {
    };
}

namespace B {
    namespace C {
        using namespace A;
    }
    typedef C::Foo FooClass;
}

namespace BIGB = B;

namespace D {
    using BIGB::FooClass;
    class Bar : public FooClass {
    }
};

class Spam : public D::Bar {
};

void evil(A::Foo *a, B::FooClass *b, B::C::Foo *c, BIGB::FooClass *d,
          BIGB::C::Foo *e, D::FooClass *f);
```

Given the possibility for such perversion, it's hard to imagine how every C++ programmer might want such code wrapped into the target language. Clearly this code defines three different classes. However, one of those classes is accessible under at least six different class names!

SWIG fully supports C++ namespaces in its internal type system and class handling code. If you feed SWIG the above code, it will be parsed correctly, it will generate compilable wrapper code, and it will produce a working scripting language module. However, the default wrapping behavior is to flatten namespaces in the target language. This means that the contents of all namespaces are merged together in the resulting scripting language module. For example, if you have code like this,

```
%module foo
namespace foo {
    void bar(int);
    void spam();
}

namespace bar {
```

```
void blah();
}
```

then SWIG simply creates three wrapper functions `bar()`, `spam()`, and `blah()` in the target language. SWIG does not prepend the names with a namespace prefix nor are the functions packaged in any kind of nested scope.

There is some rationale for taking this approach. Since C++ namespaces are often used to define modules in C++, there is a natural correlation between the likely contents of a SWIG module and the contents of a namespace. For instance, it would not be unreasonable to assume that a programmer might make a separate extension module for each C++ namespace. In this case, it would be redundant to prepend everything with an additional namespace prefix when the module itself already serves as a namespace in the target language. Or put another way, if you want SWIG to keep namespaces separate, simply wrap each namespace with its own SWIG interface.

Because namespaces are flattened, it is possible for symbols defined in different namespaces to generate a name conflict in the target language. For example:

```
namespace A {
    void foo(int);
}
namespace B {
    void foo(double);
}
```

When this conflict occurs, you will get an error message that resembles this:

```
example.i:26. Error. 'foo' is multiply defined in the generated module.
example.i:23. Previous declaration of 'foo'
```

To resolve this error, simply use `%rename` to disambiguate the declarations. For example:

```
%rename(B_foo) B::foo;
...
namespace A {
    void foo(int);
}
namespace B {
    void foo(double);    // Gets renamed to B_foo
}
```

Similarly, `%ignore` can be used to ignore declarations.

`using` declarations do not have any effect on the generated wrapper code. They are ignored by SWIG language modules and they do not result in any code. However, these declarations *are* used by the internal type system to track type-names. Therefore, if you have code like this:

```
namespace A {
    typedef int Integer;
}
using namespace A;
void foo(Integer x);
```

SWIG knows that `Integer` is the same as `A::Integer` which is the same as `int`.

Namespaces may be combined with templates. If necessary, the `%template` directive can be used to expand a template defined in a different namespace. For example:

```
namespace foo {
    template<typename T> max(T a, T b) { return a > b ? a : b; }
}
```



```
using foo::max;

%template(maxint)    max<int>           // Okay.
%template(maxfloat)  foo::max<float>;   // Okay (qualified name).

namespace bar {
    using namespace foo;
    %template(maxdouble)  max<double>;   // Okay.
}
```

The combination of namespaces and other SWIG directives may introduce subtle scope-related problems. The key thing to keep in mind is that all SWIG generated wrappers are produced in the *global* namespace. Symbols from other namespaces are always accessed using fully qualified names—names are never imported into the global space unless the interface happens to do so with a `using` declaration. In almost all cases, SWIG adjusts typenames and symbols to be fully qualified. However, this is not done in code fragments such as function bodies, typemaps, exception handlers, and so forth. For example, consider the following:

```
namespace foo {
    typedef int Integer;
    class bar {
    public:
        ...
    };
}

%extend foo::bar {
    Integer add(Integer x, Integer y) {
        Integer r = x + y;           // Error. Integer not defined in this scope
        return r;
    }
};
```

In this case, SWIG correctly resolves the added method parameters and return type to `foo::Integer`. However, since function bodies aren't parsed and such code is emitted in the global namespace, this code produces a compiler error about `Integer`. To fix the problem, make sure you use fully qualified names. For example:

```
%extend foo::bar {
    Integer add(Integer x, Integer y) {
        foo::Integer r = x + y;       // Ok.
        return r;
    }
};
```

**Note:** SWIG does *not* propagate `using` declarations to the resulting wrapper code. If these declarations appear in an interface, they should *also* appear in any header files that might have been included in a `%{ ... %}` section. In other words, don't insert extra `using` declarations into a SWIG interface unless they also appear in the underlying C++ code.

**Note:** Code inclusion directives such as `%{ ... %}` or `%inline %{ ... %}` should not be placed inside a namespace declaration. The code emitted by these directives will not be enclosed in a namespace and you may get very strange results. If you need to use namespaces with these directives, consider the following:

```
// Good version
%inline %{
namespace foo {
    void bar(int) { ... }
    ...
}
%}

// Bad version.  Emitted code not placed in namespace.
namespace foo {
%inline %{
    void bar(int) { ... }    /* I'm bad */
    ...
}
```

```
%}
}
```

**Note:** When the `%extend` directive is used inside a namespace, the namespace name is included in the generated functions. For example, if you have code like this,

```
namespace foo {
    class bar {
    public:
        %extend {
            int blah(int x);
        };
    };
}
```

the added method `blah()` is mapped to a function `int foo_bar_blah(foo::bar *self, int x)`. This function resides in the global namespace.

**Note:** Although namespaces are flattened in the target language, the SWIG generated wrapper code observes the same namespace conventions as used in the input file. Thus, if there are no symbol conflicts in the input, there will be no conflicts in the generated code.

**Note:** Namespaces have a subtle effect on the wrapping of conversion operators. For instance, suppose you had an interface like this:

```
namespace foo {
    class bar;
    class spam {
    public:
        ...
        operator bar();      // Conversion of spam -> bar
        ...
    };
}
```

To wrap the conversion function, you might be inclined to write this:

```
%rename(tofoo) foo::spam::operator bar();
```

The only problem is that it doesn't work. The reason it doesn't work is that `bar` is not defined in the global scope. Therefore, to make it work, do this instead:

```
%rename(tofoo) foo::spam::operator foo::bar();
```

**Note:** The flattening of namespaces is only intended to serve as a basic namespace implementation. Since namespaces are a new addition to SWIG, none of the target language modules are currently programmed with any namespace awareness. In the future, language modules may or may not provide more advanced namespace support.

## 6.20 Exception specifiers

When C++ programs utilize exceptions, exceptional behavior is sometimes specified as part of a function or method declaration. For example:

```
class Error { };

class Foo {
public:
    ...
    void blah() throw(Error);
    ...
};
```

If an exception specification is used, SWIG automatically generates wrapper code for catching the indicated exception and converting it into an error in the target language. In certain language modules, wrapped exception classes themselves can be used to catch errors. For example, in Python, you can write code like this:

```
f = Foo()
try:
    f.blah()
except Error,e:
    # e is a wrapped instance of "Error"
```

Obviously, the exact details of how exceptions are handled depend on the target language module.

Since exception specifiers are sometimes only used sparingly, this alone may not be enough to properly handle C++ exceptions. To do that, a different set of special SWIG directives are used. Consult the "[Customization features](#)" chapter for details.

## 6.21 Pointers to Members

Starting with SWIG1.3.7, there is limited parsing support for pointers to C++ class members. For example:

```
double do_op(Object *o, double (Object::*callback)(double,double));
extern double (Object::*fooptr)(double,double);
%constant double (Object::*FOO)(double,double) = &Object::foo;
```

Although these kinds of pointers can be parsed and represented by the SWIG type system, few language modules know how to handle them due to implementation differences from standard C pointers. Readers are *strongly* advised to consult an advanced text such as the "The Annotated C++ Manual" for specific details.

When pointers to members are supported, the pointer value might appear as a special string like this:

```
>>> print example.FOO
_ff0d54a800000000_m_Object__f_double_double__double
>>>
```

In this case, the hexadecimal digits represent the entire value of the pointer which is usually the contents of a small C++ structure on most machines.

SWIG's type-checking mechanism is also more limited when working with member pointers. Normally SWIG tries to keep track of inheritance when checking types. However, no such support is currently provided for member pointers.

## 6.22 Smart pointers and operator->()

In some C++ programs, objects are often encapsulated by smart-pointers or proxy classes. This is sometimes done to implement automatic memory management (reference counting) or persistence. Typically a smart-pointer is defined by a template class where the `->` operator has been overloaded. This class is then wrapped around some other class. For example:

```
// Smart-pointer class
template<class T> class SmartPtr {
    T *pointee;
public:
    ...
    T *operator->() {
        return pointee;
    }
    ...
};

// Ordinary class
class Foo_Impl {
public:
    int x;
```

```

    virtual void bar();
    ...
};

// Smart-pointer wrapper
typedef SmartPtr<Foo_Impl> Foo;

// Create smart pointer Foo
Foo make_Foo() {
    return SmartPtr(new Foo_Impl());
}

// Do something with smart pointer Foo
void do_something(Foo f) {
    printf("x = %d\n", f->x);
    f->bar();
}

```

A key feature of this approach is that by defining `operator->` the methods and attributes of the object wrapped by a smart pointer are transparently accessible. For example, expressions such as these (from the previous example),

```

f->x
f->bar()

```

are transparently mapped to the following

```

(f.operator->())->x;
(f.operator->())->bar();

```

When generating wrappers, SWIG tries to emulate this functionality to the extent that it is possible. To do this, whenever `operator->()` is encountered in a class, SWIG looks at its returned type and uses it to generate wrappers for accessing attributes of the underlying object. For example, wrapping the above code produces wrappers like this:

```

int Foo_x_get(Foo *f) {
    return (*f)->x;
}
void Foo_x_set(Foo *f, int value) {
    (*f)->x = value;
}
void Foo_bar(Foo *f) {
    (*f)->bar();
}

```

These wrappers take a smart-pointer instance as an argument, but dereference it in a way to gain access to the object returned by `operator->()`. You should carefully compare these wrappers to those in the first part of this chapter (they are slightly different).

The end result is that access looks very similar to C++. For example, you could do this in Python:

```

>>> f = make_Foo()
>>> print f.x
0
>>> f.bar()
>>>

```

When generating wrappers through a smart-pointer, SWIG tries to generate wrappers for all methods and attributes that might be accessible through `operator->()`. This includes any methods that might be accessible through inheritance. However, there are a number of restrictions:

- Only member variables and methods are wrapped through a smart pointer. Static members, enumerations, constructors, and destructors are not wrapped.

- If the smart-pointer class and the underlying object both define a method or variable of the same name, then the smart-pointer version has precedence. For example, if you have this code

```
class Foo {
public:
    int x;
};

class Bar {
public:
    int x;
    Foo *operator->();
};
```

then the wrapper for `Bar::x` accesses the `x` defined in `Bar`, and not the `x` defined in `Foo`.

If your intent is to only expose the smart-pointer class in the interface, it is not necessary to wrap both the smart-pointer class and the class for the underlying object. However, you must still tell SWIG about both classes if you want the technique described in this section to work. To only generate wrappers for the smart-pointer class, you can use the `%ignore` directive. For example:

```
%ignore Foo;
class Foo {          // Ignored
};

class Bar {
public:
    Foo *operator->();
    ...
};
```

Alternatively, you can import the definition of `Foo` from a separate file using `%import`.

**Note:** When a class defines `operator->()`, the operator itself is wrapped as a method `__deref__()`. For example:

```
f = Foo()           # Smart-pointer
p = f.__deref__()   # Raw pointer from operator->
```

**Note:** To disable the smart-pointer behavior, use `%ignore` to ignore `operator->()`. For example:

```
%ignore Bar::operator->;
```

**Note:** Smart pointer support was first added in SWIG-1.3.14.

## 6.23 Using declarations and inheritance

using declarations are sometimes used to adjust access to members of base classes. For example:

```
class Foo {
public:
    int  blah(int x);
};

class Bar {
public:
    double blah(double x);
};

class FooBar : public Foo, public Bar {
public:
    using Foo::blah;
    using Bar::blah;
    char *blah(const char *x);
};
```

```
};
```

In this example, the `using` declarations make different versions of the overloaded `blah()` method accessible from the derived class. For example:

```
FooBar *f;
f->blah(3);           // Ok. Invokes Foo::blah(int)
f->blah(3.5);         // Ok. Invokes Bar::blah(double)
f->blah("hello");     // Ok. Invokes FooBar::blah(const char *);
```

SWIG emulates the same functionality when creating wrappers. For example, if you wrap this code in Python, the module works just like you would expect:

```
>>> import example
>>> f = example.FooBar()
>>> f.blah(3)
>>> f.blah(3.5)
>>> f.blah("hello")
```

`using` declarations can also be used to change access when applicable. For example:

```
class Foo {
protected:
    int x;
    int blah(int x);
};

class Bar : public Foo {
public:
    using Foo::x;           // Make x public
    using Foo::blah;       // Make blah public
};
```

This also works in SWIG—the exposed declarations will be wrapped normally.

When `using` declarations are used as shown in these examples, declarations from the base classes are copied into the derived class and wrapped normally. When copied, the declarations retain any properties that might have been attached using `%rename`, `%ignore`, or `%feature`. Thus, if a method is ignored in a base class, it will also be ignored by a `using` declaration.

Because a `using` declaration does not provide fine-grained control over the declarations that get imported, it may be difficult to manage such declarations in applications that make heavy use of SWIG customization features. If you can't get `using` to work correctly, you can always change the interface to the following:

```
class FooBar : public Foo, public Bar {
public:
#ifdef SWIG
    using Foo::blah;
    using Bar::blah;
#else
    int blah(int x);           // explicitly tell SWIG about other declarations
    double blah(double x);
#endif

    char *blah(const char *x);
};
```

#### Notes:

- If a derived class redefines a method defined in a base class, then a `using` declaration won't cause a conflict. For example:

```

class Foo {
public:
    int blah(int );
    double blah(double);
};

class Bar : public Foo {
public:
    using Foo::blah;    // Only imports blah(double);
    int blah(int);
};

```

- Resolving ambiguity in overloading may prevent declarations from being imported by `using`. For example:

```

%rename(blah_long) Foo::blah(long);
class Foo {
public:
    int blah(int);
    long blah(long); // Renamed to blah_long
};

class Bar : public Foo {
public:
    using Foo::blah;    // Only imports blah(int)
    double blah(double x);
};

```

## 6.24 Partial class definitions

Since SWIG is still limited in its support of C++, it may be necessary to use partial class information in an interface file. However, since SWIG does not need the entire class specification to work, conditional compilation can be used to comment out problematic parts. For example, if you had a nested class definition, you might do this:

```

class Foo {
public:
#ifdef SWIG
    class Bar {
public:
        ...
    };
#endif
    Foo();
    ~Foo();
    ...
};

```

Also, as a rule of thumb, SWIG should not be used on raw C++ source files.

## 6.25 A brief rant about const-correctness

A common issue when working with C++ programs is dealing with all possible ways in which the `const` qualifier (or lack thereof) will break your program, all programs linked against your program, and all programs linked against those programs.

Although SWIG knows how to correctly deal with `const` in its internal type system and it knows how to generate wrappers that are free of `const`-related warnings, SWIG does not make any attempt to preserve `const`-correctness in the target language. Thus, it is possible to pass `const` qualified objects to non-`const` methods and functions. For example, consider the following code in C++:

```

const Object * foo();
void bar(Object *);

...
// C++ code

```

```
void blah() {
    bar(foo());           // Error: bar discards const
};
```

Now, consider the behavior when wrapped into a Python module:

```
>>> bar(foo())           # Okay
>>>
```

Although this is clearly a violation of the C++ type-system, fixing the problem doesn't seem to be worth the added implementation complexity that would be required to support it in the SWIG run-time type system. There are no plans to change this in future releases (although we'll never rule anything out entirely).

The bottom line is that this particular issue does not appear to be a problem for most SWIG projects. Of course, you might want to consider using another tool if maintaining constness is the most important part of your project.

## 6.26 Proxy classes

In order to provide a more natural API, SWIG's target languages wrap C++ classes with special proxy classes. These proxy classes are typically implemented in the target language itself. For example, if you're building a Python module, each C++ class is wrapped by a Python class. Or if you're building a Java module, each C++ class is wrapped by a Java class.

### 6.26.1 Construction of proxy classes

Proxy classes are always constructed as an extra layer of wrapping that uses the low-level accessor functions described in the previous section. To illustrate, suppose you had a C++ class like this:

```
class Foo {
public:
    Foo();
    ~Foo();
    int  bar(int x);
    int  x;
};
```

Using C++ as pseudocode, a proxy class looks something like this:

```
class FooProxy {
private:
    Foo    *self;
public:
    FooProxy() {
        self = new_Foo();
    }
    ~FooProxy() {
        delete_Foo(self);
    }
    int bar(int x) {
        return Foo_bar(self,x);
    }
    int x_get() {
        return Foo_x_get(self);
    }
    void x_set(int x) {
        Foo_x_set(self,x);
    }
};
```

Of course, always keep in mind that the real proxy class is written in the target language. For example, in Python, the proxy might look roughly like this:



```

class Foo:
    def __init__(self):
        self.this = new_Foo()
    def __del__(self):
        delete_Foo(self.this)
    def bar(self,x):
        return Foo_bar(self.this,x)
    def __getattr__(self,name):
        if name == 'x':
            return Foo_x_get(self.this)
        ...
    def __setattr__(self,name,value):
        if name == 'x':
            Foo_x_set(self.this,value)
        ...

```

Again, it's important to emphasize that the low-level accessor functions are always used to construct the proxy classes.

Whenever possible, proxies try to take advantage of language features that are similar to C++. This might include operator overloading, exception handling, and other features.

### 6.26.2 Resource management in proxies

A major issue with proxies concerns the memory management of wrapped objects. Consider the following C++ code:

```

class Foo {
public:
    Foo();
    ~Foo();
    int bar(int x);
    int x;
};

class Spam {
public:
    Foo *value;
    ...
};

```

Now, consider some script code that uses these classes:

```

f = Foo()           # Creates a new Foo
s = Spam()          # Creates a new Spam
s.value = f         # Stores a reference to f inside s
g = s.value         # Returns stored reference
g = 4               # Reassign g to some other value
del f              # Destroy f

```

Now, ponder the resulting memory management issues. When objects are created in the script, the objects are wrapped by newly created proxy classes. That is, there is both a new proxy class instance and a new instance of the underlying C++ class. In this example, both `f` and `s` are created in this way. However, the statement `s.value` is rather curious—when executed, a pointer to `f` is stored inside another object. This means that the scripting proxy class *AND* another C++ class share a reference to the same object. To make matters even more interesting, consider the statement `g = s.value`. When executed, this creates a new proxy class `g` that provides a wrapper around the C++ object stored in `s.value`. In general, there is no way to know where this object came from—it could have been created by the script, but it could also have been generated internally. In this particular example, the assignment of `g` results in a second proxy class for `f`. In other words, a reference to `f` is now shared by two proxy classes *and* a C++ class.

Finally, consider what happens when objects are destroyed. In the statement, `g=4`, the variable `g` is reassigned. In many languages, this makes the old value of `g` available for garbage collection. Therefore, this causes one of the proxy classes to be destroyed. Later on, the statement `del f` destroys the other proxy class. Of course, there is still a reference to the original object stored inside another C++ object. What happens to it? Is it the object still valid?

To deal with memory management problems, proxy classes always provide an API for controlling ownership. In C++ pseudocode, ownership control might look roughly like this:

```
class FooProxy {
public:
    Foo    *self;
    int    thisown;

    FooProxy() {
        self = new_Foo();
        thisown = 1;           // Newly created object
    }
    ~FooProxy() {
        if (thisown) delete_Foo(self);
    }
    ...
    // Ownership control API
    void disown() {
        thisown = 0;
    }
    void acquire() {
        thisown = 1;
    }
};

class FooPtrProxy: public FooProxy {
public:
    FooPtrProxy(Foo *s) {
        self = s;
        thisown = 0;
    }
};

class SpamProxy {
    ...
    FooProxy *value_get() {
        return FooPtrProxy(Spam_value_get(self));
    }
    void value_set(FooProxy *v) {
        Spam_value_set(self, v->self);
        v->disown();
    }
    ...
};
```

Looking at this code, there are a few central features:

- Each proxy class keeps an extra flag to indicate ownership. C++ objects are only destroyed if the ownership flag is set.
- When new objects are created in the target language, the ownership flag is set.
- When a reference to an internal C++ object is returned, it is wrapped by a proxy class, but the proxy class does not have ownership.
- In certain cases, ownership is adjusted. For instance, when a value is assigned to the member of a class, ownership is lost.
- Manual ownership control is provided by special `disown()` and `acquire()` methods.

Given the tricky nature of C++ memory management, it is impossible for proxy classes to automatically handle every possible memory management problem. However, proxies do provide a mechanism for manual control that can be used (if necessary) to address some of the more tricky memory management problems.

### 6.26.3 Language specific details

Language specific details on proxy classes are contained the chapters describing each target language. This chapter has merely introduced the topic in a very general way.

## 6.27 Where to go for more information

If you're wrapping serious C++ code, you might want to pick up a copy of "The Annotated C++ Reference Manual" by Ellis and Stroustrup. This is the reference document we use to guide a lot of SWIG's C++ support.

# 7 Preprocessing

- [File inclusion](#)
- [File imports](#)
- [Conditional Compilation](#)
- [Macro Expansion](#)
- [SWIG Macros](#)
- [C99 and GNU Extensions](#)
- [Preprocessing and `%{ ... }` blocks](#)
- [Preprocessing and `{ ... }`](#)
- [Viewing preprocessor output](#)

SWIG includes its own enhanced version of the C preprocessor. The preprocessor supports the standard preprocessor directives and macro expansion rules. However, a number of modifications and enhancements have been made. This chapter describes some of these modifications.

## 7.1 File inclusion

To include another file into a SWIG interface, use the `%include` directive like this:

```
%include "pointer.i"
```

Unlike, `#include`, `%include` includes each file once (and will not reload the file on subsequent `%include` declarations). Therefore, it is not necessary to use include-guards in SWIG interfaces.

By default, the `#include` is ignored unless you run SWIG with the `-includeall` option. The reason for ignoring traditional includes is that you often don't want SWIG to try and wrap everything included in standard header system headers and auxilliary files.

## 7.2 File imports

SWIG provides another file inclusion directive with the `%import` directive. For example:

```
%import "foo.i"
```

The purpose of `%import` is to collect certain information from another SWIG interface file or a header file without actually generating any wrapper code. Such information generally includes type declarations (e.g., `typedef`) as well as C++ classes that might be used as base-classes for class declarations in the interface. The use of `%import` is also important when SWIG is used to generate extensions as a collection of related modules. This is an advanced topic and is described in a later chapter.

The `-importall` directive tells SWIG to follow all `#include` statements as imports. This might be useful if you want to extract type definitions from system header files without generating any wrappers.

## 7.3 Conditional Compilation

SWIG fully supports the use of `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif` to conditionally include parts of an interface. The following symbols are predefined by SWIG when it is parsing the interface:

SWIG	Always defined when SWIG is processing a file
SWIGIMPORTED	Defined when SWIG is importing a file with <code>%import</code>
SWIGMAC	Defined when running SWIG on the Macintosh
SWIGWIN	Defined when running SWIG under Windows
SWIG_VERSION	Hexadecimal number containing SWIG version, such as 0x010311 (corresponding to SWIG-1.3.11).
SWIGCHICKEN	Defined when using CHICKEN

## SWIG-1.3 Documentation

SWIGCSHARP	Defined when using C#
SWIGGUILE	Defined when using Guile
SWIGJAVA	Defined when using Java
SWIGMZSCHEME	Defined when using Mzscheme
SWIGOCAML	Defined when using Ocaml
SWIGPERL	Defined when using Perl
SWIGPERL5	Defined when using Perl5
SWIGPHP	Defined when using PHP
SWIGPHP4	Defined when using PHP4
SWIGPYTHON	Defined when using Python
SWIGRUBY	Defined when using Ruby
SWIGSEXP	Defined when using S-expressions
SWIGTCL	Defined when using Tcl
SWIGTCL8	Defined when using Tcl8.0
SWIGXML	Defined when using XML

In addition, SWIG defines the following set of standard C/C++ macros:

<code>__LINE__</code>	Current line number
<code>__FILE__</code>	Current file name
<code>__STDC__</code>	Defined to indicate ANSI C
<code>__cplusplus</code>	Defined when <code>-c++</code> option used

Interface files can look at these symbols as necessary to change the way in which an interface is generated or to mix SWIG directives with C code. These symbols are also defined within the C code generated by SWIG (except for the symbol ``SWIG'` which is only defined within the SWIG compiler).

## 7.4 Macro Expansion

Traditional preprocessor macros can be used in SWIG interfaces. Be aware that the `#define` statement is also used to try and detect constants. Therefore, if you have something like this in your file,

```
#ifndef _FOO_H 1
#define _FOO_H 1
...
#endif
```

you may get some extra constants such as `_FOO_H` showing up in the scripting interface.

More complex macros can be defined in the standard way. For example:

```
#define EXTERN extern
#ifdef __STDC__
#define _ANSI(args) (args)
#else
#define _ANSI(args) ()
#endif
```

The following operators can appear in macro definitions:

- `#x`  
Converts macro argument `x` to a string surrounded by double quotes ("`x`").
- `x ## y`  
Concatenates `x` and `y` together to form `xy`.
- ``x``  
If `x` is a string surrounded by double quotes, do nothing. Otherwise, turn into a string like `#x`. This is a non-standard SWIG extension.

## 7.5 SWIG Macros

SWIG provides an enhanced macro capability with the `%define` and `%enddef` directives. For example:

```
%define ARRAYHELPER(type,name)
%inline %{
type *new_ ## name (int nitems) {
    return (type *) malloc(sizeof(type)*nitems);
}
void delete_ ## name(type *t) {
    free(t);
}
type name ## _get(type *t, int index) {
    return t[index];
}
void name ## _set(type *t, int index, type val) {
    t[index] = val;
}
%}
%enddef

ARRAYHELPER(int, IntArray)
ARRAYHELPER(double, DoubleArray)
```

The primary purpose of `%define` is to define large macros of code. Unlike normal C preprocessor macros, it is not necessary to terminate each line with a continuation character (`\`)—the macro definition extends to the first occurrence of `%enddef`. Furthermore, when such macros are expanded, they are reparsed through the C preprocessor. Thus, SWIG macros can contain all other preprocessor directives except for nested `%define` statements.

The SWIG macro capability is a very quick and easy way to generate large amounts of code. In fact, many of SWIG's advanced features and libraries are built using this mechanism (such as C++ template support).

## 7.6 C99 and GNU Extensions

SWIG-1.3.12 and newer releases support variadic preprocessor macros. For example:

```
#define DEBUGF(fmt,...)    fprintf(stderr,fmt, __VA_ARGS__)
```

When used, any extra arguments to `...` are placed into the special variable `__VA_ARGS__`. This also works with special SWIG macros defined using `%define`.

SWIG allows a variable number of arguments to be empty. However, this often results in an extra comma (,) and syntax error in the resulting expansion. For example:

```
DEBUGF("hello");    --> fprintf(stderr,"hello",);
```

To get rid of the extra comma, use `##` like this:

```
#define DEBUGF(fmt,...)    fprintf(stderr,fmt, ##__VA_ARGS__)
```

SWIG also supports GNU-style variadic macros. For example:

```
#define DEBUGF(fmt, args...)    fprintf(stdout,fmt,args)
```

**Comment:** It's not entirely clear how variadic macros might be useful to interface building. However, they are used internally to implement a number of SWIG directives and are provided to make SWIG more compatible with C99 code.

## 7.7 Preprocessing and %{ ... %} blocks

The SWIG preprocessor does not process any text enclosed in a code block `%{ ... %}`. Therefore, if you write code like this,

```
%{
#ifdef NEED_BLAH
int blah() {
    ...
}
#endif
%}
```

the contents of the `%{ ... %}` block are copied without modification to the output (including all preprocessor directives).

## 7.8 Preprocessing and { ... }

SWIG always runs the preprocessor on text appearing inside `{ ... }`. However, sometimes it is desirable to make a preprocessor directive pass through to the output file. For example:

```
%extend Foo {
    void bar() {
        #ifdef DEBUG
            printf("I'm in bar\n");
        #endif
    }
}
```

By default, SWIG will interpret the `#ifdef DEBUG` statement. However, if you really wanted that code to actually go into the wrapper file, prefix the preprocessor directives with `%` like this:

```
%extend Foo {
    void bar() {
        %#ifdef DEBUG
            printf("I'm in bar\n");
        %#endif
    }
}
```

SWIG will strip the extra `%` and leave the preprocessor directive in the code.

## 7.9 Viewing preprocessor output

Like many compilers, SWIG supports a `-E` command line option to display the output from the preprocessor. When the `-E` switch is used, SWIG will not generate any wrappers. Instead the results after the preprocessor has run are displayed. This might be useful as an aid to debugging and viewing the results of macro expansions.

## 8 SWIG library

- [The %include directive and library search path](#)
- [C Arrays and Pointers](#)
  - ◆ [cpointer.i](#)
  - ◆ [carrays.i](#)
  - ◆ [cmalloc.i](#)
  - ◆ [cdata.i](#)
- [C String Handling](#)
  - ◆ [Default string handling](#)
  - ◆ [Passing binary data](#)
  - ◆ [Using %newobject to release memory](#)
  - ◆ [cstring.i](#)
- [C++ Library](#)
  - ◆ [std\\_string.i](#)
  - ◆ [std\\_vector.i](#)
- [Utility Libraries](#)
  - ◆ [exception.i](#)

To help build extension modules, SWIG is packaged with a library of support files that you can include in your own interfaces. These files often define new SWIG directives or provide utility functions that can be used to access parts of the standard C and C++ libraries. This chapter provides a reference to the current set of supported library files.

**Compatibility note:** Older versions of SWIG included a number of library files for manipulating pointers, arrays, and other structures. Most these files are now deprecated and have been removed from the distribution. Alternative libraries provide similar functionality. Please read this chapter carefully if you used the old libraries.

### 8.1 The %include directive and library search path

Library files are included using the `%include` directive. When searching for files, directories are searched in the following order:

- The current directory
- Directories specified with the `-I` command line option
- `./swig_lib`
- `/usr/local/lib/swig_lib` (or wherever you installed SWIG)
- On Windows, SWIG also looks for the library relative to the location of `swig.exe`.

Within each directory, SWIG first looks for a subdirectory corresponding to a target language (e.g., `python`, `tcl`, etc.). If found, SWIG will search the language specific directory first. This allows for language-specific implementations of library files.

You can override the location of the SWIG library by setting the `SWIG_LIB` environment variable.

### 8.2 C Arrays and Pointers

This section describes library modules for manipulating low-level C arrays and pointers. The primary use of these modules is in supporting C declarations that manipulate bare pointers such as `int *`, `double *`, or `void *`. The modules can be used to allocate memory, manufacture pointers, dereference memory, and wrap pointers as class-like objects. Since these functions provide direct access to memory, their use is potentially unsafe and you should exercise caution.

#### 8.2.1 cpointer.i

The `cpointer.i` module defines macros that can be used to generate wrappers around simple C pointers. The primary use of this module is in generating pointers to primitive datatypes such as `int` and `double`.



**%pointer\_functions(type,name)**

Generates a collection of four functions for manipulating a pointer type `*`:

```
type *new_name()
```

Creates a new object of type `type` and returns a pointer to it. In C, the object is created using `calloc()`. In C++, `new` is used.

```
type *copy_name(type value)
```

Creates a new object of type `type` and returns a pointer to it. An initial value is set by copying it from `value`. In C, the object is created using `calloc()`. In C++, `new` is used.

```
type *delete_name(type *obj)
```

Deletes an object type `type`.

```
void name_assign(type *obj, type value)
```

Assigns `*obj = value`.

```
type name_value(type *obj)
```

Returns the value of `*obj`.

When using this macro, `type` may be any type and `name` must be a legal identifier in the target language. `name` should not correspond to any other name used in the interface file.

Here is a simple example of using `%pointer_functions()`:

```
%module example
#include "cpointer.i"

/* Create some functions for working with "int *" */
%pointer_functions(int, intp);

/* A function that uses an "int *" */
void add(int x, int y, int *result);
```

Now, in Python:

```
>>> import example
>>> c = example.new_intp()      # Create an "int" for storing result
>>> example.add(3,4,c)         # Call function
>>> example.intp_value(c)      # Dereference
7
>>> example.delete_intp(c)     # Delete
```

**%pointer\_class(type,name)**

Wraps a pointer of type `*` inside a class-based interface. This interface is as follows:

```
struct name {
    name();                          // Create pointer object
    ~name();                         // Delete pointer object
    void assign(type value);         // Assign value
    type value();                   // Get value
    type *cast();                   // Cast the pointer to original type
    static name *frompointer(type *); // Create class wrapper from existing
```

```

// pointer
};

```

When using this macro, `type` is restricted to a simple type name like `int`, `float`, or `Foo`. Pointers and other complicated types are not allowed. `name` must be a valid identifier not already in use. When a pointer is wrapped as a class, the "class" may be transparently passed to any function that expects the pointer.

If the target language does not support proxy classes, the use of this macro will produce the example same functions as `%pointer_functions()` macro.

It should be noted that the class interface does introduce a new object or wrap a pointer inside a special structure. Instead, the raw pointer is used directly.

Here is the same example using a class instead:

```

%module example
#include "cpointer.i"

/* Wrap a class interface around an "int *" */
%pointer_class(int, intp);

/* A function that uses an "int *" */
void add(int x, int y, int *result);

```

Now, in Python (using proxy classes)

```

>>> import example
>>> c = example.intp()           # Create an "int" for storing result
>>> example.add(3,4,c)           # Call function
>>> c.value()                    # Dereference
7

```

Of the two macros, `%pointer_class` is probably the most convenient when working with simple pointers. This is because the pointers are access like objects and they can be easily garbage collected (destruction of the pointer object destroys the underlying object).

### `%pointer_cast(type1, type2, name)`

Creates a casting function that converts `type1` to `type2`. The name of the function is `name`. For example:

```

%pointer_cast(int *, unsigned int *, int_to_uint);

```

In this example, the function `int_to_uint()` would be used to cast types in the target language.

**Note:** None of these macros can be used to safely work with strings (`char *` or `char **`).

**Note:** When working with simple pointers, `typemaps` can often be used to provide more seamless operation.

## 8.2.2 carrays.i

This module defines macros that assist in wrapping ordinary C pointers as arrays. The module does not provide any safety or an extra layer of wrapping—it merely provides functionality for creating, destroying, and modifying the contents of raw C array data.

### `%array_functions(type, name)`

Creates four functions.

```

type *new_name(int nelements)

```

## SWIG-1.3 Documentation

Creates a new array of objects of type `type`. In C, the array is allocated using `calloc()`. In C++, `new []` is used.

```
type *delete_name(type *ary)
```

Deletes an array. In C, `free()` is used. In C++, `delete []` is used.

```
type name_getitem(type *ary, int index)
```

Returns the value `ary[index]`.

```
void name_setitem(type *ary, int index, type value)
```

Assigns `ary[index] = value`.

When using this macro, `type` may be any type and `name` must be a legal identifier in the target language. `name` should not correspond to any other name used in the interface file.

Here is an example of `%array_functions()`. Suppose you had a function like this:

```
void print_array(double x[10]) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("[%d] = %g\n", i, x[i]);
    }
}
```

To wrap it, you might write this:

```
%module example

#include "carrays.i"
%array_functions(double, doubleArray);

void print_array(double x[10]);
```

Now, in a scripting language, you might write this:

```
a = new_doubleArray(10)           # Create an array
for i in range(0,10):
    doubleArray_setitem(a,i,2*i)  # Set a value
print_array(a)                   # Pass to C
delete_doubleArray(a)            # Destroy array
```

### **%array\_class(type,name)**

Wraps a pointer of type `*type` inside a class-based interface. This interface is as follows:

```
struct name {
    name(int nelements);           // Create an array
    ~name();                       // Delete array
    type getitem(int index);       // Return item
    void setitem(index, type value); // Set item
    type *cast();                 // Cast to original type
    static name *frompointer(type *); // Create class wrapper from
                                    // existing pointer
};
```

When using this macro, `type` is restricted to a simple type name like `int` or `float`. Pointers and other complicated types are not allowed. `name` must be a valid identifier not already in use. When a pointer is

wrapped as a class, it can be transparently passed to any function that expects the pointer.

When combined with proxy classes, the `%array_class()` macro can be especially useful. For example:

```
%module example
#include "carrays.i"
%array_class(double, doubleArray);

void print_array(double x[10]);
```

Allows you to do this:

```
import example
c = example.doubleArray(10) # Create double[10]
for i in range(0,10):
    c[i] = 2*i               # Assign values
example.print_array(c)      # Pass to C
```

**Note:** These macros do not encapsulate C arrays inside a special data structure or proxy. There is no bounds checking or safety of any kind. If you want this, you should consider using a special array object rather than a bare pointer.

**Note:** `%array_functions()` and `%array_class()` should not be used with types of `char` or `char *`.

### 8.2.3 `cmalloc.i`

This module defines macros for wrapping the low-level C memory allocation functions `malloc()`, `calloc()`, `realloc()`, and `free()`.

**`%malloc(type [,name=type])`**

Creates a wrapper around `malloc()` with the following prototype:

```
type *malloc_name(int nbytes = sizeof(type));
```

If `type` is `void`, then the size parameter `nbytes` is required. The name parameter only needs to be specified when wrapping a type that is not a valid identifier (e.g., `"int *"`, `"double **"`, etc.).

**`%calloc(type [,name=type])`**

Creates a wrapper around `calloc()` with the following prototype:

```
type *calloc_name(int nobj =1, int sz = sizeof(type));
```

If `type` is `void`, then the size parameter `sz` is required.

**`%realloc(type [,name=type])`**

Creates a wrapper around `realloc()` with the following prototype:

```
type *realloc_name(type *ptr, int nitems);
```

Note: unlike the C `realloc()`, the wrapper generated by this macro implicitly includes the size of the corresponding type. For example, `realloc_int(p, 100)` reallocates `p` so that it holds 100 integers.

**`%free(type [,name=type])`**

Creates a wrapper around `free()` with the following prototype:

```
void free_name(type *ptr);
```

```
%sizeof(type [,name=type])
```

Creates the constant:

```
%constant int sizeof_name = sizeof(type);
```

```
%allocators(type [,name=type])
```

Generates wrappers for all five of the above operations.

Here is a simple example that illustrates the use of these macros:

```
// SWIG interface
%module example
#include "cmalloc.i"

%malloc(int);
%free(int);

%malloc(int *, intp);
%free(int *, intp);

%allocators(double);
```

Now, in a script:

```
>>> from example import *
>>> a = malloc_int()
>>> a
'_000efa70_p_int'
>>> free_int(a)
>>> b = malloc_intp()
>>> b
'_000efb20_p_p_int'
>>> free_intp(b)
>>> c = calloc_double(50)
>>> c
'_000fab98_p_double'
>>> c = realloc_double(100000)
>>> free_double(c)
>>> print sizeof_double
8
>>>
```

## 8.2.4 cdata.i

The `cdata.i` module defines functions for converting raw C data to and from strings in the target language. The primary applications of this module would be packing/unpacking of binary data structures—for instance, if you needed to extract data from a buffer. The target language must support strings with embedded binary data in order for this to work.

```
char *cdata(void *ptr, int nbytes)
```

Converts `nbytes` of data at `ptr` into a string. `ptr` can be any pointer.

```
void memmove(void *ptr, char *s)
```

Copies all of the string data in `s` into the memory pointed to by `ptr`. The string may contain embedded NULL bytes. The length of the string is implicitly determined in the underlying wrapper code.

One use of these functions is packing and unpacking data from memory. Here is a short example:

```
// SWIG interface
%module example
#include "carrays.i"
#include "cdata.i"

%array_class(int, intArray);
```

Python example:

```
>>> a = intArray(10)
>>> for i in range(0,10):
...     a[i] = i
>>> b = cdata(a,40)
>>> b
'\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00\x04
\x00\x00\x00\x05\x00\x00\x00\x06\x00\x00\x00\x07\x00\x00\x00\x08\x00\x00\x00\t'
>>> c = intArray(10)
>>> memmove(c,b)
>>> print c[4]
4
>>>
```

Since the size of data is not always known, the following macro is also defined:

**%cdata(type [,name=type])**

Generates the following function for extracting C data for a given type.

```
char *cdata_name(int nitems)
```

`nitems` is the number of items of the given type to extract.

**Note:** These functions provide direct access to memory and can be used to overwrite data. Clearly they are unsafe.

## 8.3 C String Handling

A common problem when working with C programs is dealing with functions that manipulate raw character data using `char *`. In part, problems arise because there are different interpretations of `char *`—it could be a NULL-terminated string or it could point to binary data. Moreover, functions that manipulate raw strings may mutate data, perform implicit memory allocations, or utilize fixed-sized buffers.

The problems (and perils) of using `char *` are well-known. However, SWIG is not in the business of enforcing morality. The modules in this section provide basic functionality for manipulating raw C strings.

### 8.3.1 Default string handling

Suppose you have a C function with this prototype:

```
char *foo(char *s);
```

The default wrapping behavior for this function is to set `s` to a raw `char *` that refers to the internal string data in the target language. In other words, if you were using a language like Tcl, and you wrote this,

```
% foo Hello
```

then `s` would point to the representation of "Hello" inside the Tcl interpreter. When returning a `char *`, SWIG assumes that it is a NULL-terminated string and makes a copy of it. This gives the target language its own copy of the result.

There are obvious problems with the default behavior. First, since a `char *` argument points to data inside the target language, it is **NOT** safe for a function to modify this data (doing so may corrupt the interpreter and lead to a crash). Furthermore, the default behavior does not work well with binary data. Instead, strings are assumed to be NULL-terminated.

### 8.3.2 Passing binary data

If you have a function that expects binary data,

```
int parity(char *str, int len, int initial);
```

you can wrap the parameters `(char *str, int len)` as a single argument using a `typemap`. Just do this:

```
%apply (char *STRING, int LENGTH) { (char *str, int len) };
...
int parity(char *str, int len, int initial);
```

Now, in the target language, you can use binary string data like this:

```
>>> s = "H\x00\x15eg\x09\x20"
>>> parity(s,0)
```

In the wrapper function, the passed string will be expanded to a pointer and length parameter.

### 8.3.3 Using %newobject to release memory

If you have a function that allocates memory like this,

```
char *foo() {
    char *result = (char *) malloc(...);
    ...
    return result;
}
```

then the SWIG generated wrappers will have a memory leak—the returned data will be copied into a string object and the old contents ignored.

To fix the memory leak, use the `%newobject` directive.

```
%newobject foo;
...
char *foo();
```

This will release the result.

### 8.3.4 cstring.i

The `cstring.i` library file provides a collection of macros for dealing with functions that either mutate string arguments or which try to output string data through their arguments. An example of such a function might be this rather questionable implementation:

```
void get_path(char *s) {
    // Potential buffer overflow---uh, oh.
    sprintf(s,"%s/%s", base_directory, sub_directory);
}
...
// Somewhere else in the C program
{
    char path[1024];
    ...
    get_path(path);
}
```

```
    ...
}
```

(Off topic rant: If your program really has functions like this, you would be well-advised to replace them with safer alternatives involving bounds checking).

The macros defined in this module all expand to various combinations of typemaps. Therefore, the same pattern matching rules and ideas apply.

### **%cstring\_bounded\_output(parm, maxsize)**

Turns parameter *parm* into an output value. The output string is assumed to be NULL-terminated and smaller than *maxsize* characters. Here is an example:

```
%cstring_bounded_output(char *path, 1024);
...
void get_path(char *path);
```

In the target language:

```
>>> get_path()
/home/beazley/packages/Foo/Bar
>>>
```

Internally, the wrapper function allocates a small buffer (on the stack) of the requested size and passes it as the pointer value. Data stored in the buffer is then returned as a function return value. If the function already returns a value, then the return value and the output string are returned together (multiple return values). **If more than *maxsize* bytes are written, your program will crash with a buffer overflow!**

### **%cstring\_chunk\_output(parm, chunksize)**

Turns parameter *parm* into an output value. The output string is always *chunksize* and may contain binary data. Here is an example:

```
%cstring_chunk_output(char *packet, PACKETSIZE);
...
void get_packet(char *packet);
```

In the target language:

```
>>> get_packet()
'\xa9Y:\xf6\xd7\xe1\x87\xdbH;y\x97\xf"\xd3\x99\x14V\xec\x06\xea\xa2\x88'
>>>
```

This macro is essentially identical to `%cstring_bounded_output`. The only difference is that the result is always *chunksize* characters. Furthermore, the result can contain binary data. **If more than *maxsize* bytes are written, your program will crash with a buffer overflow!**

### **%cstring\_bounded\_mutable(parm, maxsize)**

Turns parameter *parm* into a mutable string argument. The input string is assumed to be NULL-terminated and smaller than *maxsize* characters. The output string is also assumed to be NULL-terminated and less than *maxsize* characters.

```
%cstring_bounded_mutable(char *ustr, 1024);
...
void make_upper(char *ustr);
```

In the target language:



```
>>> make_upper("hello world")
'HELLO WORLD'
>>>
```

Internally, this macro is almost exactly the same as `%cstring_bounded_output`. The only difference is that the parameter accepts an input value that is used to initialize the internal buffer. It is important to emphasize that this function does not mutate the string value passed—instead it makes a copy of the input value, mutates it, and returns it as a result. **If more than *maxsize* bytes are written, your program will crash with a buffer overflow!**

### `%cstring_mutable(parm [, expansion])`

Turns parameter *parm* into a mutable string argument. The input string is assumed to be NULL-terminated. An optional parameter *expansion* specifies the number of extra characters by which the string might grow when it is modified. The output string is assumed to be NULL-terminated and less than the size of the input string plus any expansion characters.

```
%cstring_mutable(char *ustr);
...
void make_upper(char *ustr);

%cstring_mutable(char *hstr, HEADER_SIZE);
...
void attach_header(char *hstr);
```

In the target language:

```
>>> make_upper("hello world")
'HELLO WORLD'
>>> attach_header("Hello world")
'header: Hello world'
>>>
```

This macro differs from `%cstring_bounded_mutable( )` in that a buffer is dynamically allocated (on the heap using `malloc/new`). This buffer is always large enough to store a copy of the input value plus any expansion bytes that might have been requested. It is important to emphasize that this function does not directly mutate the string value passed—instead it makes a copy of the input value, mutates it, and returns it as a result. **If the function expands the result by more than *expansion* extra bytes, then the program will crash with a buffer overflow!**

### `%cstring_output_maxsize(parm, maxparm)`

This macro is used to handle bounded character output functions where both a `char *` and a maximum length parameter are provided. As input, a user simply supplies the maximum length. The return value is assumed to be a NULL-terminated string.

```
%cstring_output_maxsize(char *path, int maxpath);
...
void get_path(char *path, int maxpath);
```

In the target language:

```
>>> get_path(1024)
'/home/beazley/Packages/Foo/Bar'
>>>
```

This macro provides a safer alternative for functions that need to write string data into a buffer. User supplied buffer size is used to dynamically allocate memory on heap. Results are placed into that buffer and returned as a string object.

**%cstring\_output\_withsize(parm, maxparm)**

This macro is used to handle bounded character output functions where both a `char *` and a pointer `int *` are passed. Initially, the `int *` parameter points to a value containing the maximum size. On return, this value is assumed to contain the actual number of bytes. As input, a user simply supplies the maximum length. The output value is a string that may contain binary data.

```
%cstring_output_withsize(char *data, int *maxdata);
...
void get_data(char *data, int *maxdata);
```

In the target language:

```
>>> get_data(1024)
'x627388912'
>>> get_data(1024)
'xyzzy'
>>>
```

This macro is a somewhat more powerful version of `%cstring_output_chunk( )`. Memory is dynamically allocated and can be arbitrary large. Furthermore, a function can control how much data is actually returned by changing the value of the `maxparm` argument.

**%cstring\_output\_allocate(parm, release)**

This macro is used to return strings that are allocated within the program and returned in a parameter of type `char **`. For example:

```
void foo(char **s) {
    *s = (char *) malloc(64);
    sprintf(*s, "Hello world\n");
}
```

The returned string is assumed to be NULL-terminated. *release* specifies how the allocated memory is to be released (if applicable). Here is an example:

```
%cstring_output_allocate(char **s, free(*$1));
...
void foo(char **s);
```

In the target language:

```
>>> foo()
'Hello world\n'
>>>
```

**%cstring\_output\_allocate\_size(parm, szparm, release)**

This macro is used to return strings that are allocated within the program and returned in two parameters of type `char **` and `int *`. For example:

```
void foo(char **s, int *sz) {
    *s = (char *) malloc(64);
    *sz = 64;
    // Write some binary data
    ...
}
```

The returned string may contain binary data. *release* specifies how the allocated memory is to be released (if applicable). Here is an example:

```
%cstring_output_allocate_size(char **s, int *slen, free(*$1));
...
void foo(char **s, int *slen);
```

In the target language:

```
>>> foo()
'\xa9Y:\xf6\xd7\xe1\x87\xdbH;y\x97\x7f"\xd3\x99\x14V\xec\x06\xea\xa2\x88'
>>>
```

This is the safest and most reliable way to return binary string data in SWIG. If you have functions that conform to another prototype, you might consider wrapping them with a helper function. For example, if you had this:

```
char *get_data(int *len);
```

You could wrap it with a function like this:

```
void my_get_data(char **result, int *len) {
    *result = get_data(len);
}
```

### Comments:

- Support for the `cstring.i` module depends on the target language. Not all SWIG modules currently support this library.
- Reliable handling of raw C strings is a delicate topic. There are many ways to accomplish this in SWIG. This library provides support for a few common techniques.
- If used in C++, this library uses `new` and `delete []` for memory allocation. If using ANSI C, the library uses `malloc()` and `free()`.
- Rather than manipulating `char *` directly, you might consider using a special string structure or class instead.

## 8.4 C++ Library

The library modules in this section provide access to parts of the standard C++ library. All of these modules are new in SWIG-1.3.12 and are only the beginning phase of more complete C++ library support including support for the STL.

### 8.4.1 `std_string.i`

The `std_string.i` library provides typemaps for converting C++ `std::string` objects to and from strings in the target scripting language. For example:

```
%module example
#include "std_string.i"

std::string foo();
void bar(const std::string &x);
```

In the target language:

```
x = foo();           # Returns a string object
bar("Hello World");  # Pass string as std::string
```

This module only supports types `std::string` and `const std::string &`. Pointers and non-const references are left unmodified and returned as SWIG pointers.

This library file is fully aware of C++ namespaces. If you export `std::string` or rename it with a typedef, make sure you include those declarations in your interface. For example:

```
%module example
```

```
%include "std_string.i"

using namespace std;
typedef std::string String;
...
void foo(string s, const String &t);    // std_string typemaps still applied
```

**Note:** The `std_string` library is incompatible with Perl on some platforms. We're looking into it.

## 8.4.2 `std_vector.i`

The `std_vector.i` library provides support for the C++ `vector` class in the STL. Using this library involves the use of the `%template` directive. All you need to do is to instantiate different versions of `vector` for the types that you want to use. For example:

```
%module example
%include "std_vector.i"

namespace std {
    %template(vectori) vector<int>;
    %template(vectord) vector<double>;
};
```

When a template `vector<X>` is instantiated a number of things happen:

- A class that exposes the C++ API is created in the target language. This can be used to create objects, invoke methods, etc. This class is currently a subset of the real STL `vector` class.
- Input typemaps are defined for `vector<X>`, `const vector<X> &`, and `const vector<X> *`. For each of these, a pointer `vector<X> *` may be passed or a native list object in the target language.
- An output typemap is defined for `vector<X>`. In this case, the values in the vector are expanded into a list object in the target language.
- For all other variations of the type, the wrappers expect to receive a `vector<X> *` object in the usual manner.
- An exception handler for `std::out_of_range` is defined.
- Optionally, special methods for indexing, item retrieval, slicing, and element assignment may be defined. This depends on the target language.

To illustrate the use of this library, consider the following functions:

```
/* File : example.h */

#include <vector>
#include <algorithm>
#include <functional>
#include <numeric>

double average(std::vector<int> v) {
    return std::accumulate(v.begin(), v.end(), 0.0) / v.size();
}

std::vector<double> half(const std::vector<double> &v) {
    std::vector<double> w(v);
    for (unsigned int i=0; i<w.size(); i++)
        w[i] /= 2.0;
    return w;
}

void halve_in_place(std::vector<double> &v) {
    std::transform(v.begin(), v.end(), v.begin(),
        std::bind2nd(std::divides<double>(), 2.0));
}
```

To wrap with SWIG, you might write the following:

```
%module example
%{
#include "example.h"
%}

#include "std_vector.i"
// Instantiate templates used by example
namespace std {
    %template(IntVector) vector<int>;
    %template(DoubleVector) vector<double>;
}

// Include the header file with above prototypes
#include "example.h"
```

Now, to illustrate the behavior in the scripting interpreter, consider this Python example:

```
>>> from example import *
>>> iv = IntVector(4)           # Create an vector<int>
>>> for i in range(0,4):
...     iv[i] = i
>>> average(iv)                # Call method
1.5
>>> average([0,1,2,3])         # Call with list
1.5
>>> half([1,2,3])              # Half a list
(0.5,1.0,1.5)
>>> halve_in_place([1,2,3])    # Oops
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Type error. Expected _p_std__vectorTdouble_t
>>> dv = DoubleVector(4)
>>> for i in range(0,4):
...     dv[i] = i
>>> halve_in_place(dv)         # Ok
>>> for i in dv:
...     print i
...
0.0
0.5
1.0
1.5
>>> dv[20] = 4.5
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "example.py", line 81, in __setitem__
    def __setitem__(*args): return apply(examplec.DoubleVector__setitem__,args)
IndexError: vector index out of range
>>>
```

This library module is fully aware of C++ namespaces. If you use vectors with other names, make sure you include the appropriate using or typedef directives. For example:

```
%include "std_vector.i"

namespace std {
    %template(IntVector) vector<int>;
}

using namespace std;
typedef std::vector Vector;

void foo(vector<int> *x, const Vector &x);
```

**Note:** This module makes use of several advanced SWIG features including templated typemaps and template partial

specialization. If you are trying to wrap other C++ code with templates, you might look at the code contained in `std_vector.i`. Alternatively, you can show them the code if you want to make their head explode.

**Note:** This module is defined for all SWIG target languages. However argument conversion details and the public API exposed to the interpreter vary.

**Note:** `std_vector.i` was written by Luigi "The Amazing" Ballabio.

## 8.5 Utility Libraries

### 8.5.1 exception.i

The `exception.i` library provides a language-independent function for raising a run-time exception in the target language.

**SWIG\_exception(int code, const char \*message)**

Raises an exception in the target language. `code` is one of the following symbolic constants:

```

SWIG_MemoryError
SWIG_IOError
SWIG_RuntimeError
SWIG_IndexError
SWIG_TypeError
SWIG_DivisionByZero
SWIG_OverflowError
SWIG_SyntaxError
SWIG_ValueError
SWIG_SystemError

```

`message` is a string indicating more information about the problem.

The primary use of this module is in writing language-independent exception handlers. For example:

```

#include "exception.i"
%exception std::vector::getitem {
    try {
        $action
    } catch (std::out_of_range& e) {
        SWIG_exception(SWIG_IndexError, const_cast<char*>(e.what()));
    }
}

```

## 9 Argument Handling

- [The `typemaps.i` library](#)
  - ◆ [Introduction](#)
  - ◆ [Input parameters](#)
  - ◆ [Output parameters](#)
  - ◆ [Input/Output parameters](#)
  - ◆ [Using different names](#)
- [Applying constraints to input values](#)
  - ◆ [Simple constraint example](#)
  - ◆ [Constraint methods](#)
  - ◆ [Applying constraints to new datatypes](#)

**Disclaimer:** This chapter is under construction.

In Chapter 3, SWIG's treatment of basic datatypes and pointers was described. In particular, primitive types such as `int` and `double` are mapped to corresponding types in the target language. For everything else, pointers are used to refer to structures, classes, arrays, and other user-defined datatypes. However, in certain applications it is desirable to change SWIG's handling of a specific datatype. For example, you might want to return multiple values through the arguments of a function. This chapter describes some of the techniques for doing this.

### 9.1 The `typemaps.i` library

This section describes the `typemaps.i` library file—commonly used to change certain properties of argument conversion.

#### 9.1.1 Introduction

Suppose you had a C function like this:

```
void add(double a, double b, double *result) {
    *result = a + b;
}
```

From reading the source code, it is clear that the function is storing a value in the `double *result` parameter. However, since SWIG does not examine function bodies, it has no way to know that this is the underlying behavior.

One way to deal with this is to use the `typemaps.i` library file and write interface code like this:

```
// Simple example using typemaps
%module example
#include "typemaps.i"

%apply double *OUTPUT { double *result };
extern void add(double a, double b, double *result);
```

The `%apply` directive tells SWIG that you are going to apply a special type handling rule to a type. The `"double *OUTPUT"` specification is the name of a rule that defines how to return an output value from an argument of type `double *`. This rule gets applied to all of the datatypes listed in curly braces—in this case `"double *result"`.

When the resulting module is created, you can now use the function like this (shown for Python):

```
>>> a = add(3,4)
>>> print a
7
>>>
```

In this case, you can see how the output value normally returned in the third argument has magically been transformed into a function return value. Clearly this makes the function much easier to use since it is no longer necessary to manufacture a special `double * object` and pass it to the function somehow.

Once a typemap has been applied to a type, it stays in effect for all future occurrences of the type and name. For example, you could write the following:

```
%module example
#include "typemaps.i"

%apply double *OUTPUT { double *result };
extern void add(double a, double b, double *result);
extern void sub(double a, double b, double *result);
extern void mul(double a, double b, double *result);
extern void div(double a, double b, double *result);
...
```

In this case, the `double *OUTPUT` rule is applied to all of the functions that follow.

Typemap transformations can even be extended to multiple return values. For example, consider this code:

```
%include "typemaps.i"
%apply int *OUTPUT { int *width, int *height };

// Returns a pair (width,height)
void getwinsize(int winid, int *width, int *height);
```

In this case, the function returns multiple values, allowing it to be used like this:

```
>>> w,h = genwinsize(wid)
>>> print w
400
>>> print h
300
>>>
```

It should also be noted that although the `%apply` directive is used to associate typemap rules to datatypes, you can also use the rule names directly in arguments. For example, you could write this:

```
// Simple example using typemaps
%module example
#include "typemaps.i"

extern void add(double a, double b, double *OUTPUT);
```

Typemaps stay in effect until they are explicitly deleted or redefined to something else. To clear a typemap, the `%clear` directive should be used. For example:

```
%clear double *result;      // Remove all typemaps for double *result
```

## 9.1.2 Input parameters

The following typemaps instruct SWIG that a pointer really only holds a single input value:

```
int *INPUT
short *INPUT
long *INPUT
unsigned int *INPUT
unsigned short *INPUT
unsigned long *INPUT
double *INPUT
float *INPUT
```



When used, it allows values to be passed instead of pointers. For example, consider this function:

```
double add(double *a, double *b) {
    return *a+*b;
}
```

Now, consider this SWIG interface:

```
%module example
#include "typemaps.i"
...
extern double add(double *INPUT, double *INPUT);
```

When the function is used in the scripting language interpreter, it will work like this:

```
result = add(3,4)
```

### 9.1.3 Output parameters

The following typemap rules tell SWIG that pointer is the output value of a function. When used, you do not need to supply the argument when calling the function. Instead, one or more output values are returned.

```
int *OUTPUT
short *OUTPUT
long *OUTPUT
unsigned int *OUTPUT
unsigned short *OUTPUT
unsigned long *OUTPUT
double *OUTPUT
float *OUTPUT
```

These methods can be used as shown in an earlier example. For example, if you have this C function :

```
void add(double a, double b, double *c) {
    *c = a+b;
}
```

A SWIG interface file might look like this :

```
%module example
#include "typemaps.i"
...
extern void add(double a, double b, double *OUTPUT);
```

In this case, only a single output value is returned, but this is not a restriction. An arbitrary number of output values can be returned by applying the output rules to more than one argument (as shown previously).

If the function also returns a value, it is returned along with the argument. For example, if you had this:

```
extern int foo(double a, double b, double *OUTPUT);
```

The function will return two values like this:

```
iresult, dresult = foo(3.5, 2)
```

## 9.1.4 Input/Output parameters

When a pointer serves as both an input and output value you can use the following typemaps :

```
int *INOUT
short *INOUT
long *INOUT
unsigned int *INOUT
unsigned short *INOUT
unsigned long *INOUT
double *INOUT
float *INOUT
```

A C function that uses this might be something like this:

```
void negate(double *x) {
    *x = -(*x);
}
```

To make x function as both an input and output value, declare the function like this in an interface file :

```
%module example
#include typemaps.i
...
extern void negate(double *INOUT);
```

Now within a script, you can simply call the function normally :

```
a = negate(3);          # a = -3 after calling this
```

One subtle point of the INOUT rule is that many scripting languages enforce mutability constraints on primitive objects (meaning that simple objects like integers and strings aren't supposed to change). Because of this, you can't just modify the object's value in place as the underlying C function does in this example. Therefore, the INOUT rule returns the modified value as a new object rather than directly overwriting the value of the original input object.

**Compatibility note :** The INOUT rule used to be known as BOTH in earlier versions of SWIG. Backwards compatibility is preserved, but deprecated.

## 9.1.5 Using different names

As previously shown, the %apply directive can be used to apply the INPUT, OUTPUT, and INOUT typemaps to different argument names. For example:

```
// Make double *result an output value
%apply double *OUTPUT { double *result };

// Make Int32 *in an input value
%apply int *INPUT { Int32 *in };

// Make long *x inout
%apply long *INOUT { long *x};
```

To clear a rule, the %clear directive is used:

```
%clear double *result;
%clear Int32 *in, long *x;
```

Typemap declarations are lexically scoped so a typemap takes effect from the point of definition to the end of the file or a matching `%clear` declaration.

## 9.2 Applying constraints to input values

In addition to changing the handling of various input values, it is also possible to use typemaps to apply constraints. For example, maybe you want to insure that a value is positive, or that a pointer is non-NULL. This can be accomplished including the `constraints.i` library file.

### 9.2.1 Simple constraint example

The constraints library is best illustrated by the following interface file :

```
// Interface file with constraints
%module example
#include "constraints.i"

double exp(double x);
double log(double POSITIVE);           // Allow only positive values
double sqrt(double NONNEGATIVE);       // Non-negative values only
double inv(double NONZERO);            // Non-zero values
void free(void *NONNULL);              // Non-NULL pointers only
```

The behavior of this file is exactly as you would expect. If any of the arguments violate the constraint condition, a scripting language exception will be raised. As a result, it is possible to catch bad values, prevent mysterious program crashes and so on.

### 9.2.2 Constraint methods

The following constraints are currently available

POSITIVE	Any number > 0 (not zero)
NEGATIVE	Any number < 0 (not zero)
NONNEGATIVE	Any number >= 0
NONPOSITIVE	Any number <= 0
NONZERO	Nonzero number
NONNULL	Non-NULL pointer (pointers only).

### 9.2.3 Applying constraints to new datatypes

The constraints library only supports the primitive C datatypes, but it is easy to apply it to new datatypes using `%apply`. For example :

```
// Apply a constraint to a Real variable
%apply Number POSITIVE { Real in };

// Apply a constraint to a pointer type
%apply Pointer NONNULL { Vector * };
```

The special types of "Number" and "Pointer" can be applied to any numeric and pointer variable type respectively. To later remove a constraint, the `%clear` directive can be used :

```
%clear Real in;
%clear Vector *;
```

# 10 Typemaps

- [Introduction](#)
  - ◆ [Type conversion](#)
  - ◆ [Typemaps](#)
  - ◆ [Pattern matching](#)
  - ◆ [Reusing typemaps](#)
  - ◆ [What can be done with typemaps?](#)
  - ◆ [What can't be done with typemaps?](#)
  - ◆ [The rest of this chapter](#)
- [Typemap specifications](#)
  - ◆ [Defining a typemap](#)
  - ◆ [Typemap scope](#)
  - ◆ [Copying a typemap](#)
  - ◆ [Deleting a typemap](#)
  - ◆ [Placement of typemaps](#)
- [Pattern matching rules](#)
  - ◆ [Basic matching rules](#)
  - ◆ [Typedef reductions](#)
  - ◆ [Default typemaps](#)
  - ◆ [Mixed default typemaps](#)
  - ◆ [Multi-arguments typemaps](#)
- [Code generation rules](#)
  - ◆ [Scope](#)
  - ◆ [Declaring new local variables](#)
  - ◆ [Special variables](#)
- [Common typemap methods](#)
  - ◆ ["in" typemap](#)
  - ◆ ["typecheck" typemap](#)
  - ◆ ["out" typemap](#)
  - ◆ ["arginit" typemap](#)
  - ◆ ["default" typemap](#)
  - ◆ ["check" typemap](#)
  - ◆ ["argout" typemap](#)
  - ◆ ["freearg" typemap](#)
  - ◆ ["newfree" typemap](#)
  - ◆ ["memberin" typemap](#)
  - ◆ ["varin" typemap](#)
  - ◆ ["varout" typemap](#)
  - ◆ ["throws" typemap](#)
- [Some typemap examples](#)
  - ◆ [Typemaps for arrays](#)
  - ◆ [Implementing constraints with typemaps](#)
- [Multi-argument typemaps](#)
- [The run-time type checker](#)
- [Typemaps and overloading](#)
- [More about %apply and %clear](#)
- [Reducing wrapper code size](#)
- [Passing data between typemaps](#)
- [Where to go for more information?](#)

**Disclaimer: This chapter is under construction!**

## 10.1 Introduction

Chances are, you are reading this chapter for one of two reasons; you either want to customize SWIG's behavior or you overheard someone mumbling some incomprehensible drivel about "typemaps" and you asked yourself "typemaps, what are those?" That said, let's start with a short disclaimer that "typemaps" are an advanced customization feature that provide direct access to SWIG's low-level code generator. Not only that, they are an integral part of the SWIG C++ type system (a non-trivial topic of its own). Typemaps are generally *not* a required part of using SWIG. Therefore, you might want to re-read the earlier chapters if you have found your way to this chapter with only a vague idea of what SWIG already does by default.

### 10.1.1 Type conversion

One of the most important problems in wrapper code generation is the conversion of datatypes between programming languages. Specifically, for every C/C++ declaration, SWIG must somehow generate wrapper code that allows values to be passed back and forth between languages. Since every programming language represents data differently, this is not a simple matter of simply linking code together with the C linker. Instead, SWIG has to know something about how data is represented in each language and how it can be manipulated.

To illustrate, suppose you had a simple C function like this:

```
int factorial(int n);
```

To access this function from Python, a pair of Python API functions are used to convert integer values. For example:

```
long PyInt_AsLong(PyObject *obj);      /* Python --> C */
PyObject *PyInt_FromLong(long x);      /* C --> Python */
```

The first function is used to convert the input argument from a Python integer object to C long. The second function is used to convert a value from C back into a Python integer object.

Inside the wrapper function, you might see these functions used like this:

```
PyObject *wrap_factorial(PyObject *self, PyObject *args) {
    int      arg1;
    int      result;
    PyObject *obj1;
    PyObject *resultobj;

    if (!PyArg_ParseTuple("O:factorial", &obj1)) return NULL;
    arg1 = PyInt_AsLong(obj1);
    result = factorial(arg1);
    resultobj = PyInt_FromLong(result);
    return resultobj;
}
```

Every target language supported by SWIG has functions that work in a similar manner. For example, in Perl, the following functions are used:

```
IV SvIV(SV *sv);          /* Perl --> C */
void sv_setiv(SV *sv, IV val); /* C --> Perl */
```

In Tcl:

```
int Tcl_GetLongFromObj(Tcl_Interp *interp, Tcl_Obj *obj, long *value);
Tcl_Obj *Tcl_NewIntObj(long value);
```

The precise details are not so important. What is important is that all of the underlying type conversion is handled by collections of utility functions and short bits of C code like this—you simply have to read the extension documentation for your favorite language to know how it works (an exercise left to the reader).

## 10.1.2 Typemaps

Since type handling is so central to wrapper code generation, SWIG allows it to be completely defined (or redefined) by the user. To do this, a special `%typemap` directive is used. For example:

```
/* Convert from Python --> C */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}

/* Convert from C --> Python */
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

At first glance, this code will look a little confusing. However, there is really not much to it. The first typemap (the "in" typemap) is used to convert a value from the target language to C. The second typemap (the "out" typemap) is used to convert in the other direction. The content of each typemap is a small fragment of C code that is inserted directly into the SWIG generated wrapper functions. Within this code, a number of special variables prefixed with a `$` are expanded. These are really just placeholders for C variables that are generated in the course of creating the wrapper function. In this case, `$input` refers to an input object that needs to be converted to C and `$result` refers to an object that is going to be returned by a wrapper function. `$1` refers to a C variable that has the same type as specified in the typemap declaration (an `int` in this example).

A short example might make this a little more clear. If you were wrapping a function like this:

```
int gcd(int x, int y);
```

A wrapper function would look approximately like this:

```
PyObject *wrap_gcd(PyObject *self, PyObject *args) {
    int arg1;
    int arg2;
    int result;
    PyObject *obj1;
    PyObject *obj2;
    PyObject *resultobj;

    if (!PyArg_ParseTuple("OO:gcd", &obj1, &obj2)) return NULL;

    /* "in" typemap, argument 1 */
    {
        arg1 = PyInt_AsLong(obj1);
    }

    /* "in" typemap, argument 2 */
    {
        arg2 = PyInt_AsLong(obj2);
    }

    result = gcd(arg1, arg2);

    /* "out" typemap, return value */
    {
        resultobj = PyInt_FromLong(result);
    }

    return resultobj;
}
```

In this code, you can see how the typemap code has been inserted into the function. You can also see how the special `$` variables have been expanded to match certain variable names inside the wrapper function. This is really the whole idea behind typemaps—they simply let you insert arbitrary code into different parts of the generated wrapper functions. Because arbitrary

code can be inserted, it possible to completely change the way in which values are converted.

### 10.1.3 Pattern matching

As the name implies, the purpose of a typemap is to "map" C datatypes to types in the target language. Once a typemap is defined for a C datatype, it is applied to all future occurrences of that type in the input file. For example:

```
/* Convert from Perl --> C */
%typemap(in) int {
    $1 = SvIV($input);
}

...
int factorial(int n);
int gcd(int x, int y);
int count(char *s, char *t, int max);
```

The matching of typemaps to C datatypes is more than a simple textual match. In fact, typemaps are fully built into the underlying type system. Therefore, typemaps are unaffected by `typedef`, namespaces, and other declarations that might hide the underlying type. For example, you could have code like this:

```
/* Convert from Ruby--> C */
%typemap(in) int {
    $1 = NUM2INT($input);
}

...
typedef int Integer;
namespace foo {
    typedef Integer Number;
};

int foo(int x);
int bar(Integer y);
int spam(foo::Number a, foo::Number b);
```

In this case, the typemap is still applied to the proper arguments even though typenames don't always match the text "int". This ability to track types is a critical part of SWIG—in fact, all of the target language modules work merely define a set of typemaps for the basic types. Yet, it is never necessary to write new typemaps for typenames introduced by `typedef`.

In addition to tracking typenames, typemaps may also be specialized to match against a specific argument name. For example, you could write a typemap like this:

```
%typemap(in) double nonnegative {
    $1 = PyFloat_AsDouble($input);
    if ($1 < 0) {
        PyErr_SetString(PyExc_ValueError, "argument must be nonnegative.");
        return NULL;
    }
}

...
double sin(double x);
double cos(double x);
double sqrt(double nonnegative);

typedef double Real;
double log(Real nonnegative);
...
```

For certain tasks such as input argument conversion, typemaps can be defined for sequences of consecutive arguments. For example:

```
%typemap(in) (char *str, int len) {
```

```

    $1 = PyString_AsString($input);    /* char *str */
    $2 = PyString_Size($input);        /* int len  */
}
...
int count(char *str, int len, char c);

```

In this case, a single input object is expanded into a pair of C arguments. This example also provides a hint to the unusual variable naming scheme involving \$1, \$2, and so forth.

### 10.1.4 Reusing typemaps

Typemaps are normally defined for specific type and argument name patterns. However, typemaps can also be copied and reused. One way to do this is to use assignment like this:

```

%typemap(in) Integer = int;
%typemap(in) (char *buffer, int size) = (char *str, int len);

```

A more general form of copying is found in the %apply directive like this:

```

%typemap(in) int {
    /* Convert an integer argument */
    ...
}
%typemap(out) int {
    /* Return an integer value */
    ...
}

/* Apply all of the integer typemaps to size_t */
%apply int { size_t };

```

%apply merely takes *all* of the typemaps that are defined for one type and applies them to other types. Note: you can include a comma separated set of types in the { ... } part of %apply.

It should be noted that it is not necessary to copy typemaps for types that are related by typedef. For example, if you have this,

```
typedef int size_t;
```

then SWIG already knows that the int typemaps apply. You don't have to do anything.

### 10.1.5 What can be done with typemaps?

The primary use of typemaps is for defining wrapper generation behavior at the level of individual C/C++ datatypes. There are currently six general categories of problems that typemaps address:

#### Argument handling

```
int foo(int x, double y, char *s);
```

- Input argument conversion ("in" typemap).
- Input argument type checking ("typecheck" typemap).
- Output argument handling ("argout" typemap).
- Input argument value checking ("check" typemap).
- Input argument initialization ("arginit" typemap).
- Default arguments ("default" typemap).
- Input argument resource management ("freearg" typemap).

#### Return value handling



```
int foo(int x, double y, char *s);
```

- Function return value conversion ("out" typemap).
- Return value resource management ("ret" typemap).
- Resource management for newly allocated objects ("newfree" typemap).

### Exception handling

```
int foo(int x, double y, char *s) throw(MemoryError, IndexError);
```

- Handling of C++ exception specifiers. ("throw" typemap).

### Global variables

```
int foo;
```

- Assignment of a global variable. ("varin" typemap).
- Reading a global variable. ("varout" typemap).

### Member variables

```
struct Foo {
    int x[20];
};
```

- Assignment of data to a class/structure member. ("memberin" typemap).

### Constant creation

```
#define FOO 3
%constant int BAR = 42;
enum { ALE, LAGER, STOUT };
```

- Creation of constant values. ("consttab" or "constcode" typemap).

Details of each of these typemaps will be covered shortly. Also, certain language modules may define additional typemaps that expand upon this list. For example, the Java module defines a variety of typemaps for controlling additional aspects of the Java bindings. Consult language specific documentation for further details.

## 10.1.6 What can't be done with typemaps?

Typemaps can't be used to define properties that apply to C/C++ declarations as a whole. For example, suppose you had a declaration like this,

```
Foo *make_Foo();
```

and you wanted to tell SWIG that `make_Foo()` returned a newly allocated object (for the purposes of providing better memory management). Clearly, this property of `make_Foo()` is *not* a property that would be associated with the datatype `Foo *` by itself. Therefore, a completely different SWIG customization mechanism (`%feature`) is used for this purpose. Consult the [Customization Features](#) chapter for more information about that.

Typemaps also can't be used to rearrange or transform the order of arguments. For example, if you had a function like this:

```
void foo(int, char *);
```

you can't use typemaps to interchange the arguments, allowing you to call the function like this:

```
foo("hello",3)           # Reversed arguments
```

If you want to change the calling conventions of a function, write a helper function instead. For example:

```
%rename(foo) wrap_foo;
%inline %{
void wrap_foo(char *s, int x) {
    foo(x,s);
}
%}
```

### 10.1.7 The rest of this chapter

The rest of this chapter provides detailed information for people who want to write new typemaps. This information is of particular importance to anyone who intends to write a new SWIG target language module. Power users can also use this information to write application specific type conversion rules.

Since typemaps are strongly tied to the underlying C++ type system, subsequent sections assume that you are reasonably familiar with the basic details of values, pointers, references, arrays, type qualifiers (e.g., `const`), structures, namespaces, templates, and memory management in C/C++. If not, you would be well-advised to consult a copy of "The C Programming Language" by Kernighan and Ritchie or "The C++ Programming Language" by Stroustrup before going any further.

## 10.2 Typemap specifications

This section describes the behavior of the `%typemap` directive itself.

### 10.2.1 Defining a typemap

New typemaps are defined using the `%typemap` declaration. The general form of this declaration is as follows (parts enclosed in `[ ... ]` are optional):

```
%typemap(method [, modifiers]) typelist code ;
```

*method* is a simply a name that specifies what kind of typemap is being defined. It is usually a name like "in", "out", or "argout". The purpose of these methods is described later.

*modifiers* is an optional comma separated list of `name="value"` values. These are sometimes to attach extra information to a typemap and is often target-language dependent.

*typelist* is a list of the C++ type patterns that the typemap will match. The general form of this list is as follows:

```
typelist      :  typepattern [, typepattern, typepattern, ... ] ;

typepattern  :  type [ (parms) ]
               |  type name [ (parms) ]
               |  ( typelist ) [ (parms) ]
```

Each type pattern is either a simple type, a simple type and argument name, or a list of types in the case of multi-argument typemaps. In addition, each type pattern can be parameterized with a list of temporary variables (*parms*). The purpose of these variables will be explained shortly.

*code* specifies the C code used in the typemap. It can take any one of the following forms:

```
code          :  { ... }
               |  " ... "
               |  %{ ... %}
```

Here are some examples of valid typemap specifications:

```

/* Simple typemap declarations */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}
%typemap(in) int "$1 = PyInt_AsLong($input)";
%typemap(in) int %{
    $1 = PyInt_AsLong($input);
}%}

/* Typemap with extra argument name */
%typemap(in) int nonnegative {
    ...
}

/* Multiple types in one typemap */
%typemap(in) int, short, long {
    $1 = SvIV($input);
}

/* Typemap with modifiers */
%typemap(in,doc="integer") int "$1 = gh_scm2int($input)";

/* Typemap applied to patterns of multiple arguments */
%typemap(in) (char *str, int len),
              (char *buffer, int size)
{
    $1 = PyString_AsString($input);
    $2 = PyString_Size($input);
}

/* Typemap with extra pattern parameters */
%typemap(in, numinputs=0) int *output (int temp),
                          long *output (long temp)
{
    $1 = &temp;
}

```

Admittedly, it's not the most readable syntax at first glance. However, the purpose of the individual pieces will become clear.

## 10.2.2 Typemap scope

Once defined, a typemap remains in effect for all of the declarations that follow. A typemap may be redefined for different sections of an input file. For example:

```

// typemap1
%typemap(in) int {
    ...
}

int fact(int);                // typemap1
int gcd(int x, int y);        // typemap1

// typemap2
%typemap(in) int {
    ...
}

int isprime(int);             // typemap2

```

One exception to the typemap scoping rules pertains to the `%extend` declaration. `%extend` is used to attach new declarations to a class or structure definition. Because of this, all of the declarations in an `%extend` block are subject to the typemap rules that are in effect at the point where the class itself is defined. For example:

```

class Foo {
    ...
}

```

```
};

%typemap(in) int {
    ...
}

%extend Foo {
    int blah(int x);    // typemap has no effect. Declaration is attached to Foo which
                        // appears before the %typemap declaration.
};
```

### 10.2.3 Copying a typemap

A typemap is copied by using assignment. For example:

```
%typemap(in) Integer = int;
```

or this:

```
%typemap(in) Integer, Number, int32_t = int;
```

Types are often managed by a collection of different typemaps. For example:

```
%typemap(in)      int { ... }
%typemap(out)     int { ... }
%typemap(varin)   int { ... }
%typemap(varout)  int { ... }
```

To copy all of these typemaps to a new type, use %apply. For example:

```
%apply int { Integer };           // Copy all int typemaps to Integer
%apply int { Integer, Number };   // Copy all int typemaps to both Integer and Number
```

The patterns for %apply follow the same rules as for %typemap. For example:

```
%apply int *output { Integer *output };           // Typemap with name
%apply (char *buf, int len) { (char *buffer, int size) }; // Multiple arguments
```

### 10.2.4 Deleting a typemap

A typemap can be deleted by simply defining no code. For example:

```
%typemap(in) int;                // Clears typemap for int
%typemap(in) int, long, short;    // Clears typemap for int, long, short
%typemap(in) int *output;
```

The %clear directive clears all typemaps for a given type. For example:

```
%clear int;                      // Removes all types for int
%clear int *output, long *output;
```

**Note:** Since SWIG's default behavior is defined by typemaps, clearing a fundamental type like `int` will make that type unusable unless you also define a new set of typemaps immediately after the clear operation.

### 10.2.5 Placement of typemaps

Typemap declarations can be declared in the global scope, within a C++ namespace, and within a C++ class. For example:

```
%typemap(in) int {
    ...
}
```

```

}

namespace std {
    class string;
    %typemap(in) string {
        ...
    }
}

class Bar {
public:
    typedef const int & const_reference;
    %typemap(out) const_reference {
        ...
    }
};

```

When a typemap appears inside a namespace or class, it stays in effect until the end of the SWIG input (just like before). However, the typemap takes the local scope into account. Therefore, this code

```

namespace std {
    class string;
    %typemap(in) string {
        ...
    }
}

```

is really defining a typemap for the type `std::string`. You could have code like this:

```

namespace std {
    class string;
    %typemap(in) string {          /* std::string */
        ...
    }
}

namespace Foo {
    class string;
    %typemap(in) string {          /* Foo::string */
        ...
    }
}

```

In this case, there are two completely distinct typemaps that apply to two completely different types (`std::string` and `Foo::string`).

It should be noted that for scoping to work, SWIG has to know that `string` is a typename defined within a particular namespace. In this example, this is done using the class declaration `class string`.

## 10.3 Pattern matching rules

The section describes the pattern matching rules by which C datatypes are associated with typemaps.

### 10.3.1 Basic matching rules

Typemaps are matched using both a type and a name (typically the name of a argument). For a given `TYPE NAME` pair, the following rules are applied, in order, to find a match. The first typemap found is used.

- Typemaps that exactly match `TYPE` and `NAME`.
- Typemaps that exactly match `TYPE` only.

If `TYPE` includes qualifiers (`const`, `volatile`, etc.), they are stripped and the following checks are made:

- Typemaps that match the stripped `TYPE` and `NAME`.
- Typemaps that match the stripped `TYPE` only.

If `TYPE` is an array. The following transformation is made:

- Replace all dimensions to `[ANY]` and look for a generic array typemap.

To illustrate, suppose that you had a function like this:

```
int foo(const char *s);
```

To find a typemap for the argument `const char *s`, SWIG will search for the following typemaps:

<code>const char *s</code>	Exact type and name match
<code>const char *</code>	Exact type match
<code>char *s</code>	Type and name match (stripped qualifiers)
<code>char *</code>	Type match (stripped qualifiers)

When more than one typemap rule might be defined, only the first match found is actually used. Here is an example that shows how some of the basic rules are applied:

```
%typemap(in) int *x {
    ... typemap 1
}

%typemap(in) int * {
    ... typemap 2
}

%typemap(in) const int *z {
    ... typemap 3
}

%typemap(in) int [4] {
    ... typemap 4
}

%typemap(in) int [ANY] {
    ... typemap 5
}

void A(int *x);           // int *x rule      (typemap 1)
void B(int *y);           // int * rule      (typemap 2)
void C(const int *x);     // int *x rule      (typemap 1)
void D(const int *z);     // int * rule      (typemap 3)
void E(int x[4]);         // int [4] rule     (typemap 4)
void F(int x[1000]);      // int [ANY] rule   (typemap 5)
```

### 10.3.2 Typedef reductions

If no match is found using the rules in the previous section, SWIG applies a typedef reduction to the type and repeats the typemap search for the reduced type. To illustrate, suppose you had code like this:

```
%typemap(in) int {
    ... typemap 1
}

typedef int Integer;
void blah(Integer x);
```

## SWIG-1.3 Documentation

To find the typemap for `Integer x`, SWIG will first search for the following typemaps:

```
Integer x
Integer
```

Finding no match, it then applies a reduction `Integer --> int` to the type and repeats the search.

```
int x
int      --> match: typemap 1
```

Even though two types might be the same via typedef, SWIG allows typemaps to be defined for each typename independently. This allows for interesting customization possibilities based solely on the typename itself. For example, you could write code like this:

```
typedef double  pdouble;      // Positive double

// typemap 1
%typemap(in) double {
    ... get a double ...
}
// typemap 2
%typemap(in) pdouble {
    ... get a positive double ...
}
double sin(double x);          // typemap 1
pdouble sqrt(pdouble x);      // typemap 2
```

When reducing the type, only one typedef reduction is applied at a time. The search process continues to apply reductions until a match is found or until no more reductions can be made.

For complicated types, the reduction process can generate a long list of patterns. Consider the following:

```
typedef int Integer;
typedef Integer Row4[4];
void foo(Row4 rows[10]);
```

To find a match for the `Row4 rows[10]` argument, SWIG would check the following patterns, stopping only when it found a match:

```
Row4 rows[10]
Row4 [10]
Row4 rows[ANY]
Row4 [ANY]

# Reduce Row4 --> Integer[4]
Integer rows[10][4]
Integer [10][4]
Integer rows[ANY][ANY]
Integer [ANY][ANY]

# Reduce Integer --> int
int rows[10][4]
int [10][4]
int rows[ANY][ANY]
int [ANY][ANY]
```

For parametrized types like templates, the situation is even more complicated. Suppose you had some declarations like this:

```
typedef int Integer;
typedef foo<Integer,Integer> fooii;
void blah(fooii *x);
```

In this case, the following typemap patterns are searched for the argument `fooii *x`:

```
fooii *x
fooii *

# Reduce fooii --> foo<Integer,Integer>
foo<Integer,Integer> *x
foo<Integer,Integer> *

# Reduce Integer -> int
foo<int, Integer> *x
foo<int, Integer> *

# Reduce Integer -> int
foo<int, int> *x
foo<int, int> *
```

Typemap reductions are always applied to the left—most type that appears. Only when no reductions can be made to the left—most type are reductions made to other parts of the type. This behavior means that you could define a typemap for `foo<int, Integer>`, but a typemap for `foo<Integer, int>` would never be matched. Admittedly, this is rather esoteric—there's little practical reason to write a typemap quite like that. Of course, you could rely on this to confuse your coworkers even more.

### 10.3.3 Default typemaps

Most SWIG language modules use typemaps to define the default behavior of the C primitive types. This is entirely straightforward. For example, a set of typemaps are written like this:

```
%typemap(in) int      "convert an int";
%typemap(in) short    "convert a short";
%typemap(in) float     "convert a float";
...
```

Since typemap matching follows all `typedef` declarations, any sort of type that is mapped to a primitive type through `typedef` will be picked up by one of these primitive typemaps.

The default behavior for pointers, arrays, references, and other kinds of types are handled by specifying rules for variations of the reserved `SWIGTYPE` type. For example:

```
%typemap(in) SWIGTYPE *      { ... default pointer handling ... }
%typemap(in) SWIGTYPE &      { ... default reference handling ... }
%typemap(in) SWIGTYPE []     { ... default array handling ... }
%typemap(in) enum SWIGTYPE   { ... default handling for enum values ... }
%typemap(in) SWIGTYPE (CLASS::*) { ... default pointer member handling ... }
```

These rules match any kind of pointer, reference, or array—even when multiple levels of indirection or multiple array dimensions are used. Therefore, if you wanted to change SWIG's default handling for all types of pointers, you would simply redefine the rule for `SWIGTYPE *`.

Finally, the following typemap rule is used to match against simple types that don't match any other rules:

```
%typemap(in) SWIGTYPE { ... handle an unknown type ... }
```

This typemap is important because it is the rule that gets triggered when `call` or `return` by value is used. For instance, if you have a declaration like this:

```
double dot_product(Vector a, Vector b);
```

The `Vector` type will usually just get matched against `SWIGTYPE`. The default implementation of `SWIGTYPE` is to convert the value into pointers (as described in chapter 3).



By redefining `SWIGTYPE` it may be possible to implement other behavior. For example, if you cleared all typemaps for `SWIGTYPE`, SWIG simply won't wrap any unknown datatype (which might be useful for debugging). Alternatively, you might modify `SWIGTYPE` to marshal objects into strings instead of converting them to pointers.

The best way to explore the default typemaps is to look at the ones already defined for a particular language module. Typemap definitions are usually found in the SWIG library in a file such as `python.swg`, `tcl8.swg`, etc.

### 10.3.4 Mixed default typemaps

The default typemaps described above can be mixed with `const` and with each other. For example the `SWIGTYPE *` typemap is for default pointer handling, but if a `const SWIGTYPE *` typemap is defined it will be used instead for constant pointers. Some further examples follow:

```
%typemap(in) enum SWIGTYPE &      { ... enum references ... }
%typemap(in) const enum SWIGTYPE & { ... const enum references ... }
%typemap(in) SWIGTYPE *&           { ... pointers passed by reference ... }
%typemap(in) SWIGTYPE * const &    { ... constant pointers passed by reference ... }
%typemap(in) SWIGTYPE[ANY][ANY]    { ... 2D arrays ... }
```

Note that the the typedef reduction described earlier is also used with these mixed default typemaps. For example, say the following typemaps are defined and SWIG is looking for the best match for the enum shown below:

```
%typemap(in) const Hello &      { ... }
%typemap(in) const enum SWIGTYPE & { ... }
%typemap(in) enum SWIGTYPE &    { ... }
%typemap(in) SWIGTYPE &         { ... }
%typemap(in) SWIGTYPE           { ... }

enum Hello {};
const Hello &hi;
```

The typemap at the top of the list will be chosen, not because it is defined first, but because it is the closest match for the type being wrapped. If any of the typemaps in the above list were not defined, then the next one on the list would have precedence. In other words the typemap chosen is the closest explicit match.

**Compatibility note:** The mixed default typemaps were introduced in SWIG-1.3.23, but were not used much in this version. Expect to see them being used more and more within the various libraries in later versions of SWIG.

### 10.3.5 Multi-arguments typemaps

When multi-argument typemaps are specified, they take precedence over any typemaps specified for a single type. For example:

```
%typemap(in) (char *buffer, int len) {
    // typemap 1
}

%typemap(in) char *buffer {
    // typemap 2
}

void foo(char *buffer, int len, int count); // (char *buffer, int len)
void bar(char *buffer, int blah);           // char *buffer
```

Multi-argument typemaps are also more restrictive in the way that they are matched. Currently, the first argument follows the matching rules described in the previous section, but all subsequent arguments must match exactly.

## 10.4 Code generation rules

This section describes rules by which typemap code is inserted into the generated wrapper code.

### 10.4.1 Scope

When a typemap is defined like this:

```
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}
```

the typemap code is inserted into the wrapper function using a new block scope. In other words, the wrapper code will look like this:

```
wrap_whatever() {
    ...
    // Typemap code
    {
        arg1 = PyInt_AsLong(obj1);
    }
    ...
}
```

Because the typemap code is enclosed in its own block, it is legal to declare temporary variables for use during typemap execution. For example:

```
%typemap(in) short {
    long temp;          /* Temporary value */
    if (Tcl_GetLongFromObj(interp, $input, &temp) != TCL_OK) {
        return TCL_ERROR;
    }
    $1 = (short) temp;
}
```

Of course, any variables that you declare inside a typemap are destroyed as soon as the typemap code has executed (they are not visible to other parts of the wrapper function or other typemaps that might use the same variable names).

Occasionally, typemap code will be specified using a few alternative forms. For example:

```
%typemap(in) int "$1 = PyInt_AsLong($input)";
%typemap(in) int %{
    $1 = PyInt_AsLong($input);
}%
```

These two forms are mainly used for cosmetics—the specified code is not enclosed inside a block scope when it is emitted. This sometimes results in a less complicated looking wrapper function.

### 10.4.2 Declaring new local variables

Sometimes it is useful to declare a new local variable that exists within the scope of the entire wrapper function. A good example of this might be an application in which you wanted to marshal strings. Suppose you had a C++ function like this

```
int foo(std::string *s);
```

and you wanted to pass a native string in the target language as an argument. For instance, in Perl, you wanted the function to work like this:

```
$x = foo("Hello World");
```

To do this, you can't just pass a raw Perl string as the `std::string *` argument. Instead, you have to create a temporary `std::string` object, copy the Perl string data into it, and then pass a pointer to the object. To do this, simply specify the typemap with an extra parameter like this:

```
%typemap(in) std::string * (std::string temp) {
    unsigned int len;
    char      *s;
    s = SvPV($input,len);      /* Extract string data */
    temp.assign(s,len);        /* Assign to temp */
    $1 = &temp;                /* Set argument to point to temp */
}
```

In this case, `temp` becomes a local variable in the scope of the entire wrapper function. For example:

```
wrap_foo() {
    std::string temp;    <--- Declaration of temp goes here
    ...

    /* Typemap code */
    {
        ...
        temp.assign(s,len);
        ...
    }
    ...
}
```

When you set `temp` to a value, it persists for the duration of the wrapper function and gets cleaned up automatically on exit.

It is perfectly safe to use more than one typemap involving local variables in the same declaration. For example, you could declare a function as :

```
void foo(std::string *x, std::string *y, std::string *z);
```

This is safely handled because SWIG actually renames all local variable references by appending an argument number suffix. Therefore, the generated code would actually look like this:

```
wrap_foo() {
    int *arg1;    /* Actual arguments */
    int *arg2;
    int *arg3;
    std::string temp1;    /* Locals declared in the typemap */
    std::string temp2;
    std::string temp3;
    ...
    {
        char *s;
        unsigned int len;
        ...
        temp1.assign(s,len);
        arg1 = *temp1;
    }
    {
        char *s;
        unsigned int len;
        ...
        temp2.assign(s,len);
        arg2 = *temp2;
    }
    {
        char *s;
        unsigned int len;
        ...
        temp3.assign(s,len);
        arg3 = *temp3;
    }
    ...
}
```

Some typemaps do not recognize local variables (or they may simply not apply). At this time, only typemaps that apply to argument conversion support this.

### 10.4.3 Special variables

Within all typemaps, the following special variables are expanded.

Variable	Meaning
<code>\$n</code>	A C local variable corresponding to type <i>n</i> in the typemap pattern.
<code>\$argnum</code>	Argument number. Only available in typemaps related to argument conversion
<code>\$n_name</code>	Argument name
<code>\$n_type</code>	Real C datatype of type <i>n</i> .
<code>\$n_ltype</code>	ltype of type <i>n</i>
<code>\$n_mangle</code>	Mangled form of type <i>n</i> . For example <code>_p_Foo</code>
<code>\$n_descriptor</code>	Type descriptor structure for type <i>n</i> . For example <code>SWIGTYPE_p_Foo</code> . This is primarily used when interacting with the run-time type checker (described later).
<code>*\$n_type</code>	Real C datatype of type <i>n</i> with one pointer removed.
<code>*\$n_ltype</code>	ltype of type <i>n</i> with one pointer removed.
<code>*\$n_mangle</code>	Mangled form of type <i>n</i> with one pointer removed.
<code>*\$n_descriptor</code>	Type descriptor structure for type <i>n</i> with one pointer removed.
<code>\$\$n_type</code>	Real C datatype of type <i>n</i> with one pointer added.
<code>\$\$n_ltype</code>	ltype of type <i>n</i> with one pointer added.
<code>\$\$n_mangle</code>	Mangled form of type <i>n</i> with one pointer added.
<code>\$\$n_descriptor</code>	Type descriptor structure for type <i>n</i> with one pointer added.
<code>\$n_basetype</code>	Base typename with all pointers and qualifiers stripped.

Within the table, `$n` refers to a specific type within the typemap specification. For example, if you write this

```
%typemap(in) int *INPUT {
}
```

then `$1` refers to `int *INPUT`. If you have a typemap like this,

```
%typemap(in) (int argc, char *argv[]) {
    ...
}
```

then `$1` refers to `int argc` and `$2` refers to `char *argv[ ]`.

Substitutions related to types and names always fill in values from the actual code that was matched. This is useful when a typemap might match multiple C datatype. For example:

```
%typemap(in) int, short, long {
    $1 = ($1_ltype) PyInt_AsLong($input);
}
```

In this case, `$1_ltype` is replaced with the datatype that is actually matched.

When typemap code is emitted, the C/C++ datatype of the special variables `$1` and `$2` is always an "ltype." An "ltype" is simply a type that can legally appear on the left-hand side of a C assignment operation. Here are a few examples of types and ltypes:

```
type          ltype
-----
```

int	int
const int	int
const int *	int *
int [4]	int *
int [4][5]	int (*)[5]

In most cases a ltype is simply the C datatype with qualifiers stripped off. In addition, arrays are converted into pointers.

Variables such as `$_l_type` and `$_l_type` are used to safely modify the type by removing or adding pointers. Although not needed in most typemaps, these substitutions are sometimes needed to properly work with typemaps that convert values between pointers and values.

If necessary, type related substitutions can also be used when declaring locals. For example:

```
%typemap(in) int * ($_l_type temp) {
    temp = PyInt_AsLong($input);
    $l = &temp;
}
```

There is one word of caution about declaring local variables in this manner. If you declare a local variable using a type substitution such as `$_l_type temp`, it won't work like you expect for arrays and certain kinds of pointers. For example, if you wrote this,

```
%typemap(in) int [10][20] {
    $_l_type temp;
}
```

then the declaration of `temp` will be expanded as

```
int (*)[20] temp;
```

This is illegal C syntax and won't compile. There is currently no straightforward way to work around this problem in SWIG due to the way that typemap code is expanded and processed. However, one possible workaround is to simply pick an alternative type such as `void *` and use casts to get the correct type when needed. For example:

```
%typemap(in) int [10][20] {
    void *temp;
    ...
    (($_l_type) temp)[i][j] = x;    /* set a value */
    ...
}
```

Another approach, which only works for arrays is to use the `$_l_basetype` substitution. For example:

```
%typemap(in) int [10][20] {
    $_l_basetype temp[10][20];
    ...
    temp[i][j] = x;    /* set a value */
    ...
}
```

## 10.5 Common typemap methods

The set of typemaps recognized by a language module may vary. However, the following typemap methods are nearly universal:

### 10.5.1 "in" typemap

The "in" typemap is used to convert function arguments from the target language to C. For example:

```
%typemap(in) int {
```

```
$1 = PyInt_AsLong($input);
}
```

The following special variables are available:

\$input	- Input object holding value to be converted.
\$symname	- Name of function/method being wrapped

This is probably the most commonly redefined typemap because it can be used to implement customized conversions.

In addition, the "in" typemap allows the number of converted arguments to be specified. For example:

```
// Ignored argument.
%typemap(in, numinputs=0) int *out (int temp) {
    $1 = &temp;
}
```

At this time, only zero or one arguments may be converted.

**Compatibility note:** Specifying numinputs=0 is the same as the old "ignore" typemap.

### 10.5.2 "typecheck" typemap

The "typecheck" typemap is used to support overloaded functions and methods. It merely checks an argument to see whether or not it matches a specific type. For example:

```
%typemap(typecheck, precedence=SWIG_TYPECHECK_INTEGER) int {
    $1 = PyInt_Check($input) ? 1 : 0;
}
```

For typechecking, the \$1 variable is always a simple integer that is set to 1 or 0 depending on whether or not the input argument is the correct type.

If you define new "in" typemaps *and* your program uses overloaded methods, you should also define a collection of "typecheck" typemaps. More details about this follow in a later section on "Typemaps and Overloading."

### 10.5.3 "out" typemap

The "out" typemap is used to convert function/method return values from C into the target language. For example:

```
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

The following special variables are available.

\$result	- Result object returned to target language.
\$symname	- Name of function/method being wrapped

### 10.5.4 "arginit" typemap

The "arginit" typemap is used to set the initial value of a function argument—before any conversion has occurred. This is not normally necessary, but might be useful in highly specialized applications. For example:

```
// Set argument to NULL before any conversion occurs
%typemap(arginit) int *data {
    $1 = NULL;
}
```

### 10.5.5 "default" typemap

The "default" typemap is used to turn an argument into a default argument. For example:

```
%typemap(default) int flags {
    $1 = DEFAULT_FLAGS;
}
...
int foo(int x, int y, int flags);
```

The primary use of this typemap is to either change the wrapping of default arguments or specify a default argument in a language where they aren't supported (like C). Target languages that do not support optional arguments, such as Java and C#, effectively ignore the value specified by this typemap as all arguments must be given.

Once a default typemap has been applied to an argument, all arguments that follow must have default values. See the [Default/optional arguments](#) section for further information on default argument wrapping.

### 10.5.6 "check" typemap

The "check" typemap is used to supply value checking code during argument conversion. The typemap is applied *after* arguments have been converted. For example:

```
%typemap(check) int positive {
    if ($1 <= 0) {
        SWIG_exception(SWIG_ValueError, "Expected positive value.");
    }
}
```

### 10.5.7 "argout" typemap

The "argout" typemap is used to return values from arguments. This is most commonly used to write wrappers for C/C++ functions that need to return multiple values. The "argout" typemap is almost always combined with an "in" typemap—possibly to ignore the input value. For example:

```
/* Set the input argument to point to a temporary variable */
%typemap(in, numinputs=0) int *out (int temp) {
    $1 = &temp;
}

%typemap(argout) int *out {
    // Append output value $1 to $result
    ...
}
```

The following special variables are available.

\$result	- Result object returned to target language.
\$input	- The original input object passed.
\$symname	- Name of function/method being wrapped

The code supplied to the "argout" typemap is always placed after the "out" typemap. If multiple return values are used, the extra return values are often appended to return value of the function.

See the `typemaps.i` library for examples.

### 10.5.8 "freearg" typemap

The "freearg" typemap is used to cleanup argument data. It is only used when an argument might have allocated resources that need to be cleaned up when the wrapper function exits. The "freearg" typemap usually cleans up argument resources allocated by

the "in" typemap. For example:

```
// Get a list of integers
%typemap(in) int *items {
    int nitems = Length($input);
    $1 = (int *) malloc(sizeof(int)*nitems);
}
// Free the list
%typemap(freearg) int *items {
    free($1);
}
```

The "freearg" typemap inserted at the end of the wrapper function, just before control is returned back to the target language. This code is also placed into a special variable `$cleanup` that may be used in other typemaps whenever a wrapper function needs to abort prematurely.

### 10.5.9 "newfree" typemap

The "newfree" typemap is used in conjunction with the `%newobject` directive and is used to deallocate memory used by the return result of a function. For example:

```
%typemap(newfree) string * {
    delete $1;
}
%typemap(out) string * {
    $result = PyString_FromString($1->c_str());
}
...

%newobject foo;
...
string *foo();
```

### 10.5.10 "memberin" typemap

The "memberin" typemap is used to copy data from *an already converted input value* into a structure member. It is typically used to handle array members and other special cases. For example:

```
%typemap(memberin) int [4] {
    memmove($1, $input, 4*sizeof(int));
}
```

It is rarely necessary to write "memberin" typemaps—SWIG already provides a default implementation for arrays, strings, and other objects.

### 10.5.11 "varin" typemap

The "varin" typemap is used to convert objects in the target language to C for the purposes of assigning to a C/C++ global variable. This is implementation specific.

### 10.5.12 "varout" typemap

The "varout" typemap is used to convert a C/C++ object to an object in the target language when reading a C/C++ global variable. This is implementation specific.

### 10.5.13 "throws" typemap

The "throws" typemap is only used when SWIG parses a C++ method with an exception specification. It provides a default mechanism for handling C++ methods that have declared the exceptions it will throw. The purpose of this typemap is to convert a



C++ exception into an error or exception in the target language. It is slightly different to the other typemaps as it is based around the exception type rather than the type of a parameter or variable. For example:

```
%typemap(throws) const char * %{
    PyErr_SetString(PyExc_RuntimeError, $1);
    SWIG_fail;
}%
void bar() throw (const char *);
```

As can be seen from the generated code below, SWIG generates an exception handler with the catch block comprising the "throws" typemap content.

```
...
try {
    bar();
}
catch(char const *_e) {
    PyErr_SetString(PyExc_RuntimeError, _e);
    SWIG_fail;
}
...
```

Note that if your methods do not have an exception specification yet they do throw exceptions, SWIG cannot know how to deal with them. For a neat way to handle these, see the [Exception handling with %exception](#) section.

## 10.6 Some typemap examples

This section contains a few examples. Consult language module documentation for more examples.

### 10.6.1 Typemaps for arrays

A common use of typemaps is to provide support for C arrays appearing both as arguments to functions and as structure members.

For example, suppose you had a function like this:

```
void set_vector(int type, float value[4]);
```

If you wanted to handle `float value[4]` as a list of floats, you might write a typemap similar to this:

```
%typemap(in) float value[4] (float temp[4]) {
    int i;
    if (!PySequence_Check($input)) {
        PyErr_SetString(PyExc_ValueError, "Expected a sequence");
        return NULL;
    }
    if (PySequence_Length($input) != 4) {
        PyErr_SetString(PyExc_ValueError, "Size mismatch. Expected 4 elements");
        return NULL;
    }
    for (i = 0; i < 4; i++) {
        PyObject *o = PySequence_GetItem($input, i);
        if (PyNumber_Check(o)) {
            temp[i] = (float) PyFloat_AsDouble(o);
        } else {
            PyErr_SetString(PyExc_ValueError, "Sequence elements must be numbers");
            return NULL;
        }
    }
    $1 = temp;
}
```

In this example, the variable `temp` allocates a small array on the C stack. The `typemap` then populates this array and passes it to the underlying C function.

When used from Python, the `typemap` allows the following type of function call:

```
>>> set_vector(type, [ 1, 2.5, 5, 20 ])
```

If you wanted to generalize the `typemap` to apply to arrays of all dimensions you might write this:

```
%typemap(in) float value[ANY] (float temp[$1_dim0]) {
    int i;
    if (!PySequence_Check($input)) {
        PyErr_SetString(PyExc_ValueError,"Expected a sequence");
        return NULL;
    }
    if (PySequence_Length($input) != $1_dim0) {
        PyErr_SetString(PyExc_ValueError,"Size mismatch. Expected $1_dim0 elements");
        return NULL;
    }
    for (i = 0; i < $1_dim0; i++) {
        PyObject *o = PySequence_GetItem($input,i);
        if (PyNumber_Check(o)) {
            temp[i] = (float) PyFloat_AsDouble(o);
        } else {
            PyErr_SetString(PyExc_ValueError,"Sequence elements must be numbers");
            return NULL;
        }
    }
    $1 = temp;
}
```

In this example, the special variable `$1_dim0` is expanded with the actual array dimensions. Multidimensional arrays can be matched in a similar manner. For example:

```
%typemap(python,in) float matrix[ANY][ANY] (float temp[$1_dim0][$1_dim1]) {
    ... convert a 2d array ...
}
```

For large arrays, it may be impractical to allocate storage on the stack using a temporary variable as shown. To work with heap allocated data, the following technique can be used.

```
%typemap(in) float value[ANY] {
    int i;
    if (!PySequence_Check($input)) {
        PyErr_SetString(PyExc_ValueError,"Expected a sequence");
        return NULL;
    }
    if (PySequence_Length($input) != $1_dim0) {
        PyErr_SetString(PyExc_ValueError,"Size mismatch. Expected $1_dim0 elements");
        return NULL;
    }
    $1 = (float *) malloc($1_dim0*sizeof(float));
    for (i = 0; i < $1_dim0; i++) {
        PyObject *o = PySequence_GetItem($input,i);
        if (PyNumber_Check(o)) {
            $1[i] = (float) PyFloat_AsDouble(o);
        } else {
            PyErr_SetString(PyExc_ValueError,"Sequence elements must be numbers");
            return NULL;
        }
    }
}
%typemap(freearg) float value[ANY] {
    if ($1) free($1);
}
```

```
}
```

In this case, an array is allocated using `malloc`. The `freearg` typemap is then used to release the argument after the function has been called.

Another common use of array typemaps is to provide support for array structure members. Due to subtle differences between pointers and arrays in C, you can't just "assign" to a array structure member. Instead, you have to explicitly copy elements into the array. For example, suppose you had a structure like this:

```
struct SomeObject {
    float  value[4];
    ...
};
```

When SWIG runs, it won't produce any code to set the `vec` member. You may even get a warning message like this:

```
swig -python example.i
Generating wrappers for Python
example.i:10. Warning. Array member value will be read-only.
```

These warning messages indicate that SWIG does not know how you want to set the `vec` field.

To fix this, you can supply a special "memberin" typemap like this:

```
%typemap(memberin) float [ANY] {
    int i;
    for (i = 0; i < $1_dim0; i++) {
        $1[i] = $input[i];
    }
}
```

The `memberin` typemap is used to set a structure member from data that has already been converted from the target language to C. In this case, `$input` is the local variable in which converted input data is stored. This typemap then copies this data into the structure.

When combined with the earlier typemaps for arrays, the combination of the "in" and "memberin" typemap allows the following usage:

```
>>> s = SomeObject()
>>> s.x = [1, 2.5, 5, 10]
```

Related to structure member input, it may be desirable to return structure members as a new kind of object. For example, in this example, you will get very odd program behavior where the structure member can be set nicely, but reading the member simply returns a pointer:

```
>>> s = SomeObject()
>>> s.x = [1, 2.5, 5, 10]
>>> print s.x
_1008fea8_p_float
>>>
```

To fix this, you can write an "out" typemap. For example:

```
%typemap(out) float [ANY] {
    int i;
    $result = PyList_New($1_dim0);
    for (i = 0; i < $1_dim0; i++) {
        PyObject *o = PyFloat_FromDouble((double) $1[i]);
        PyList_SetItem($result,i,o);
    }
}
```

Now, you will find that member access is quite nice:

```
>>> s = SomeObject()
>>> s.x = [1, 2.5, 5, 10]
>>> print s.x
[ 1, 2.5, 5, 10]
```

**Compatibility Note:** SWIG1.1 used to provide a special "memberout" typemap. However, it was mostly useless and has since been eliminated. To return structure members, simply use the "out" typemap.

## 10.6.2 Implementing constraints with typemaps

One particularly interesting application of typemaps is the implementation of argument constraints. This can be done with the "check" typemap. When used, this allows you to provide code for checking the values of function arguments. For example :

```
%module math

%typemap(check) double posdouble {
    if ($1 < 0) {
        croak("Expecting a positive number");
    }
}

...

double sqrt(double posdouble);
```

This provides a sanity check to your wrapper function. If a negative number is passed to this function, a Perl exception will be raised and your program terminated with an error message.

This kind of checking can be particularly useful when working with pointers. For example :

```
%typemap(check) Vector * {
    if ($1 == 0) {
        PyErr_SetString(PyExc_TypeError, "NULL Pointer not allowed");
        return NULL;
    }
}
```

will prevent any function involving a `Vector *` from accepting a NULL pointer. As a result, SWIG can often prevent a potential segmentation faults or other run-time problems by raising an exception rather than blindly passing values to the underlying C/C++ program.

Note: A more advanced constraint checking system is in development. Stay tuned.

## 10.7 Multi-argument typemaps

So far, the typemaps presented have focused on the problem of dealing with single values. For example, converting a single input object to a single argument in a function call. However, certain conversion problems are difficult to handle in this manner. As an example, consider the example at the very beginning of this chapter:

```
int foo(int argc, char *argv[]);
```

Suppose that you wanted to wrap this function so that it accepted a single list of strings like this:

```
>>> foo(["ale", "lager", "stout"])
```

To do this, you not only need to map a list of strings to `char *argv[]`, but the value of `int argc` is implicitly determined by the length of the list. Using only simple typemaps, this type of conversion is possible, but extremely painful. Therefore, SWIG1.3

introduces the notion of multi-argument typemaps.

A multi-argument typemap is a conversion rule that specifies how to convert a *single* object in the target language to set of consecutive function arguments in C/C++. For example, the following multi-argument maps perform the conversion described for the above example:

```
%typemap(in) (int argc, char *argv[]) {
    int i;
    if (!PyList_Check($input)) {
        PyErr_SetString(PyExc_ValueError, "Expecting a list");
        return NULL;
    }
    $1 = PyList_Size($input);
    $2 = (char **) malloc(($1+1)*sizeof(char *));
    for (i = 0; i < $1; i++) {
        PyObject *s = PyList_GetItem($input,i);
        if (!PyString_Check(s)) {
            free($2);
            PyErr_SetString(PyExc_ValueError, "List items must be strings");
            return NULL;
        }
        $2[i] = PyString_AsString(s);
    }
    $2[i] = 0;
}

%typemap(freearg) (int argc, char *argv[]) {
    if ($2) free($2);
}
```

A multi-argument map is always specified by surrounding the arguments with parentheses as shown. For example:

```
%typemap(in) (int argc, char *argv[]) { ... }
```

Within the typemap code, the variables \$1, \$2, and so forth refer to each type in the map. All of the usual substitutions apply—just use the appropriate \$1 or \$2 prefix on the variable name (e.g., \$2\_type, \$1\_ltype, etc.)

Multi-argument typemaps always have precedence over simple typemaps and SWIG always performs longest-match searching. Therefore, you will get the following behavior:

```
%typemap(in) int argc { ... typemap 1 ... }
%typemap(in) (int argc, char *argv[]) { ... typemap 2 ... }
%typemap(in) (int argc, char *argv[], char *env[]) { ... typemap 3 ... }

int foo(int argc, char *argv[]); // Uses typemap 2
int bar(int argc, int x); // Uses typemap 1
int spam(int argc, char *argv[], char *env[]); // Uses typemap 3
```

It should be stressed that multi-argument typemaps can appear anywhere in a function declaration and can appear more than once. For example, you could write this:

```
%typemap(in) (int scout, char *swords[]) { ... }
%typemap(in) (int wcount, char *words[]) { ... }

void search_words(int scout, char *swords[], int wcount, char *words[], int maxcount);
```

Other directives such as %apply and %clear also work with multi-argument maps. For example:

```
%apply (int argc, char *argv[]) {
    (int scout, char *swords[]),
    (int wcount, char *words[])
};
...
```

```
%clear (int scount, char *swords[]), (int wcount, char *words[]);
...
```

Although multi-argument typemaps may seem like an exotic, little used feature, there are several situations where they make sense. First, suppose you wanted to wrap functions similar to the low-level `read()` and `write()` system calls. For example:

```
typedef unsigned int size_t;

int read(int fd, void *rbuffer, size_t len);
int write(int fd, void *wbuffer, size_t len);
```

As is, the only way to use the functions would be to allocate memory and pass some kind of pointer as the second argument—a process that might require the use of a helper function. However, using multi-argument maps, the functions can be transformed into something more natural. For example, you might write typemaps like this:

```
// typemap for an outgoing buffer
%typemap(in) (void *wbuffer, size_t len) {
    if (!PyString_Check($input)) {
        PyErr_SetString(PyExc_ValueError, "Expecting a string");
        return NULL;
    }
    $1 = (void *) PyString_AsString($input);
    $2 = PyString_Size($input);
}

// typemap for an incoming buffer
%typemap(in) (void *rbuffer, size_t len) {
    if (!PyInt_Check($input)) {
        PyErr_SetString(PyExc_ValueError, "Expecting an integer");
        return NULL;
    }
    $2 = PyInt_AsLong($input);
    if ($2 < 0) {
        PyErr_SetString(PyExc_ValueError, "Positive integer expected");
        return NULL;
    }
    $1 = (void *) malloc($2);
}

// Return the buffer. Discarding any previous return result
%typemap(argout) (void *rbuffer, size_t len) {
    Py_XDECREF($result); /* Blow away any previous result */
    if (result < 0) { /* Check for I/O error */
        free($1);
        PyErr_SetFromErrno(PyExc_IOError);
        return NULL;
    }
    $result = PyString_FromStringAndSize($1,result);
    free($1);
}
```

(note: In the above example, `$result` and `result` are two different variables. `result` is the real C datatype that was returned by the function. `$result` is the scripting language object being returned to the interpreter.).

Now, in a script, you can write code that simply passes buffers as strings like this:

```
>>> f = example.open("Makefile")
>>> example.read(f,40)
'TOP      = ../../\nSWIG      = $(TOP)/.'
>>> example.read(f,40)
'./swig\nSRCS      = example.c\nTARGET      '
>>> example.close(f)
0
>>> g = example.open("foo", example.O_WRONLY | example.O_CREAT, 0644)
>>> example.write(g,"Hello world\n")
```

```

12
>>> example.write(g,"This is a test\n")
15
>>> example.close(g)
0
>>>
    
```

A number of multi-argument typemap problems also arise in libraries that perform matrix-calculations—especially if they are mapped onto low-level Fortran or C code. For example, you might have a function like this:

```
int is_symmetric(double *mat, int rows, int columns);
```

In this case, you might want to pass some kind of higher-level object as an matrix. To do this, you could write a multi-argument typemap like this:

```

%typemap(in) (double *mat, int rows, int columns) {
    MatrixObject *a;
    a = GetMatrixFromObject($input);      /* Get matrix somehow */

    /* Get matrix properties */
    $1 = GetPointer(a);
    $2 = GetRows(a);
    $3 = GetColumns(a);
}
    
```

This kind of technique can be used to hook into scripting-language matrix packages such as Numeric Python. However, it should also be stressed that some care is in order. For example, when crossing languages you may need to worry about issues such as row-major vs. column-major ordering (and perform conversions if needed).

## 10.8 The run-time type checker

The run-time type checker is used by many, but not all, of SWIG's supported target languages. The run-time type checker features are not required and are thus not used for strongly typed languages such as Java and C#. The scripting and scheme based languages rely on it and it forms a critical part of SWIG's operation for these languages.

When pointers, arrays, and objects are wrapped by SWIG, they are normally converted into typed pointer objects. For example, an instance of `Foo *` might be a string encoded like this:

```
_108e688_p_Foo
```

At a basic level, the type checker simply restores some type-safety to extension modules. However, the type checker is also responsible for making sure that wrapped C++ classes are handled correctly—especially when inheritance is used. This is especially important when an extension module makes use of multiple inheritance. For example:

```

class Foo {
    int x;
};

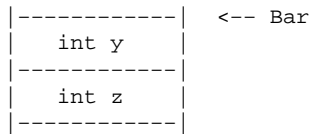
class Bar {
    int y;
};

class FooBar : public Foo, public Bar {
    int z;
};
    
```

When the class `FooBar` is organized in memory, it contains the contents of the classes `Foo` and `Bar` as well as its own data members. For example:

```

FooBar --> | ----- | <-- Foo
           |   int x   |
    
```



Because of the way that base class data is stacked together, the casting of a `FooBar *` to either of the base classes may change the actual value of the pointer. This means that it is generally not safe to represent pointers using a simple integer or a bare `void *`—type tags are needed to implement correct handling of pointer values (and to make adjustments when needed).

In the wrapper code generated for each language, pointers are handled through the use of special type descriptors and conversion functions. For example, if you look at the wrapper code for Python, you will see code like this:

```
if ((SWIG_ConvertPtr(obj0,(void **) &arg1, SWIGTYPE_p_Foo,1)) == -1) return NULL;
```

In this code, `SWIGTYPE_p_Foo` is the type descriptor that describes `Foo *`. The type descriptor is actually a pointer to a structure that contains information about the type name to use in the target language, a list of equivalent typenames (via typedef or inheritance), and pointer value handling information (if applicable). The `SWIG_ConvertPtr()` function is simply a utility function that takes a pointer object in the target language and a type-descriptor objects and uses this information to generate a C++ pointer. However, the exact name and calling conventions of the conversion function depends on the target language (see language specific chapters for details).

When pointers are converted in a `typemap`, the `typemap` code often looks similar to this:

```
%typemap(in) Foo * {
    if ((SWIG_ConvertPtr($input, (void **) &$1, $1_descriptor)) == -1) return NULL;
}
```

The most critical part is the `typemap` is the use of the `$1_descriptor` special variable. When placed in a `typemap`, this is expanded into the `SWIGTYPE_*` type descriptor object above. As a general rule, you should always use `$1_descriptor` instead of trying to hard-code the type descriptor name directly.

There is another reason why you should always use the `$1_descriptor` variable. When this special variable is expanded, SWIG marks the corresponding type as "in use." When type-tables and type information is emitted in the wrapper file, descriptor information is only generated for those datatypes that were actually used in the interface. This greatly reduces the size of the type tables and improves efficiency.

Occasionally, you might need to write a `typemap` that needs to convert pointers of other types. To handle this, a special macro substitution `$descriptor(type)` can be used to generate the SWIG type descriptor name for any C datatype. For example:

```
%typemap(in) Foo * {
    if ((SWIG_ConvertPtr($input, (void **) &$1, $1_descriptor)) == -1) {
        Bar *temp;
        if ((SWIG_ConvertPtr($input), (void **) &temp, $descriptor(Bar *)) == -1) {
            return NULL;
        }
        $1 = (Foo *) temp;
    }
}
```

The primary use of `$descriptor(type)` is when writing `typemaps` for container objects and other complex data structures. There are some restrictions on the argument—namely it must be a fully defined C datatype. It can not be any of the special `typemap` variables.

In certain cases, SWIG may not generate type-descriptors like you expect. For example, if you are converting pointers in some non-standard way or working with an unusual combination of interface files and modules, you may find that SWIG omits information for a specific type descriptor. To fix this, you may need to use the `%types` directive. For example:

```
%types(int *, short *, long *, float *, double *);
```



When `%types` is used, SWIG generates type-descriptor information even if those datatypes never appear elsewhere in the interface file.

A final problem related to the type-checker is the conversion of types in code that is external to the SWIG wrapper file. This situation is somewhat rare in practice, but occasionally a programmer may want to convert a typed pointer object into a C++ pointer somewhere else in their program. The only problem is that the SWIG type descriptor objects are only defined in the wrapper code and not normally accessible.

To correctly deal with this situation, the following technique can be used:

```
/* Some non-SWIG file */

/* External declarations */
extern void *SWIG_TypeQuery(const char *);
extern int  SWIG_ConvertPtr(PyObject *, void **ptr, void *descr);

void foo(PyObject *o) {
    Foo *f;
    static void *descr = 0;
    if (!descr) {
        descr = SWIG_TypeQuery("Foo *"); /* Get the type descriptor structure for Foo */
        assert(descr);
    }
    if ((SWIG_ConvertPtr(o, (void **) &f, descr) == -1)) {
        abort();
    }
    ...
}
```

Further details about the run-time type checking can be found in the documentation for individual language modules. Reading the source code may also help. The file `common.swg` in the SWIG library contains all of the source code for type-checking. This code is also included in every generated wrapped file so you probably just look at the output of SWIG to get a better sense for how types are managed.

## 10.9 Typemaps and overloading

In many target languages, SWIG fully supports C++ overloaded methods and functions. For example, if you have a collection of functions like this:

```
int foo(int x);
int foo(double x);
int foo(char *s, int y);
```

You can access the functions in a normal way from the scripting interpreter:

```
# Python
foo(3)           # foo(int)
foo(3.5)         # foo(double)
foo("hello",5)   # foo(char *, int)

# Tcl
foo 3            # foo(int)
foo 3.5          # foo(double)
foo hello 5      # foo(char *, int)
```

To implement overloading, SWIG generates a separate wrapper function for each overloaded method. For example, the above functions would produce something roughly like this:

```
// wrapper pseudocode
_wrap_foo_0(argc, args[]) {      // foo(int)
    int arg1;
```

```

int result;
...
arg1 = FromInteger(args[0]);
result = foo(arg1);
return ToInteger(result);
}

_wrap_foo_1(argc, args[]) {          // foo(double)
    double arg1;
    int result;
    ...
    arg1 = FromDouble(args[0]);
    result = foo(arg1);
    return ToInteger(result);
}

_wrap_foo_2(argc, args[]) {          // foo(char *, int)
    char *arg1;
    int arg2;
    int result;
    ...
    arg1 = FromString(args[0]);
    arg2 = FromInteger(args[1]);
    result = foo(arg1,arg2);
    return ToInteger(result);
}

```

Next, a dynamic dispatch function is generated:

```

_wrap_foo(argc, args[]) {
    if (argc == 1) {
        if (IsInteger(args[0])) {
            return _wrap_foo_0(argc,args);
        }
        if (IsDouble(args[0])) {
            return _wrap_foo_1(argc,args);
        }
    }
    if (argc == 2) {
        if (IsString(args[0]) && IsInteger(args[1])) {
            return _wrap_foo_2(argc,args);
        }
    }
    error("No matching function!\n");
}

```

The purpose of the dynamic dispatch function is to select the appropriate C++ function based on argument types—a task that must be performed at runtime in most of SWIG's target languages.

The generation of the dynamic dispatch function is a relatively tricky affair. Not only must input typemaps be taken into account (these typemaps can radically change the types of arguments accepted), but overloaded methods must also be sorted and checked in a very specific order to resolve potential ambiguity. A high-level overview of this ranking process is found in the "[SWIG and C++](#)" chapter. What isn't mentioned in that chapter is the mechanism by which it is implemented—as a collection of typemaps.

To support dynamic dispatch, SWIG first defines a general purpose type hierarchy as follows:

Symbolic Name	Precedence Value
-----	-----
SWIG_TYPECHECK_POINTER	0
SWIG_TYPECHECK_VOIDPTR	10
SWIG_TYPECHECK_BOOL	15
SWIG_TYPECHECK_UINT8	20
SWIG_TYPECHECK_INT8	25
SWIG_TYPECHECK_UINT16	30

SWIG_TYPECHECK_INT16	35
SWIG_TYPECHECK_UINT32	40
SWIG_TYPECHECK_INT32	45
SWIG_TYPECHECK_UINT64	50
SWIG_TYPECHECK_INT64	55
SWIG_TYPECHECK_UINT128	60
SWIG_TYPECHECK_INT128	65
SWIG_TYPECHECK_INTEGER	70
SWIG_TYPECHECK_FLOAT	80
SWIG_TYPECHECK_DOUBLE	90
SWIG_TYPECHECK_COMPLEX	100
SWIG_TYPECHECK_UNICHAR	110
SWIG_TYPECHECK_UNISTRING	120
SWIG_TYPECHECK_CHAR	130
SWIG_TYPECHECK_STRING	140
SWIG_TYPECHECK_BOOL_ARRAY	1015
SWIG_TYPECHECK_INT8_ARRAY	1025
SWIG_TYPECHECK_INT16_ARRAY	1035
SWIG_TYPECHECK_INT32_ARRAY	1045
SWIG_TYPECHECK_INT64_ARRAY	1055
SWIG_TYPECHECK_INT128_ARRAY	1065
SWIG_TYPECHECK_FLOAT_ARRAY	1080
SWIG_TYPECHECK_DOUBLE_ARRAY	1090
SWIG_TYPECHECK_CHAR_ARRAY	1130
SWIG_TYPECHECK_STRING_ARRAY	1140

(These precedence levels are defined in `swig.swg`, a library file that's included by all target language modules.)

In this table, the precedence-level determines the order in which types are going to be checked. Low values are always checked before higher values. For example, integers are checked before floats, single values are checked before arrays, and so forth.

Using the above table as a guide, each target language defines a collection of "typecheck" typemaps. The follow excerpt from the Python module illustrates this:

```
/* Python type checking rules */
/* Note: %typecheck(X) is a macro for %typemap(typecheck,precedence=X) */

%typecheck(SWIG_TYPECHECK_INTEGER)
    int, short, long,
    unsigned int, unsigned short, unsigned long,
    signed char, unsigned char,
    long long, unsigned long long,
    const int &, const short &, const long &,
    const unsigned int &, const unsigned short &, const unsigned long &,
    const long long &, const unsigned long long &,
    enum SWIGTYPE,
    bool, const bool &
{
    $1 = (PyInt_Check($input) || PyLong_Check($input)) ? 1 : 0;
}

%typecheck(SWIG_TYPECHECK_DOUBLE)
    float, double,
    const float &, const double &
{
    $1 = (PyFloat_Check($input) || PyInt_Check($input) || PyLong_Check($input)) ? 1 : 0;
}

%typecheck(SWIG_TYPECHECK_CHAR) char {
    $1 = (PyString_Check($input) && (PyString_Size($input) == 1)) ? 1 : 0;
}

%typecheck(SWIG_TYPECHECK_STRING) char * {
    $1 = PyString_Check($input) ? 1 : 0;
}
```

## SWIG-1.3 Documentation

```
%typecheck(SWIG_TYPECHECK_POINTER) SWIGTYPE *, SWIGTYPE &, SWIGTYPE [] {
    void *ptr;
    if (SWIG_ConvertPtr($input, (void **) &ptr, $l_descriptor, 0) == -1) {
        $l = 0;
        PyErr_Clear();
    } else {
        $l = 1;
    }
}

%typecheck(SWIG_TYPECHECK_POINTER) SWIGTYPE {
    void *ptr;
    if (SWIG_ConvertPtr($input, (void **) &ptr, $l_descriptor, 0) == -1) {
        $l = 0;
        PyErr_Clear();
    } else {
        $l = 1;
    }
}

%typecheck(SWIG_TYPECHECK_VOIDPTR) void * {
    void *ptr;
    if (SWIG_ConvertPtr($input, (void **) &ptr, 0, 0) == -1) {
        $l = 0;
        PyErr_Clear();
    } else {
        $l = 1;
    }
}

%typecheck(SWIG_TYPECHECK_POINTER) PyObject *
{
    $l = ($input != 0);
}
```

It might take a bit of contemplation, but this code has merely organized all of the basic C++ types, provided some simple type-checking code, and assigned each type a precedence value.

Finally, to generate the dynamic dispatch function, SWIG uses the following algorithm:

- Overloaded methods are first sorted by the number of required arguments.
- Methods with the same number of arguments are then sorted by precedence values of argument types.
- Typecheck typemaps are then emitted to produce a dispatch function that checks arguments in the correct order.

If you haven't written any typemaps of your own, it is unnecessary to worry about the typechecking rules. However, if you have written new input typemaps, you might have to supply a typechecking rule as well. An easy way to do this is to simply copy one of the existing typechecking rules. Here is an example,

```
// Typemap for a C++ string
%typemap(in) std::string {
    if (PyString_Check($input)) {
        $l = std::string(PyString_AsString($input));
    } else {
        SWIG_exception(SWIG_TypeError, "string expected");
    }
}
// Copy the typecheck code for "char *".
%typemap(typecheck) std::string = char *;
```

The bottom line: If you are writing new typemaps and you are using overloaded methods, you will probably have to write typecheck code or copy existing code. Since this is a relatively new SWIG feature, there are few examples to work with. However, you might look at some of the existing library files like 'typemaps.i' for a guide.

**Notes:**

- Typecheck typemaps are not used for non-overloaded methods. Because of this, it is still always necessary to check types in any "in" typemaps.
- The dynamic dispatch process is only meant to be a heuristic. There are many corner cases where SWIG simply can't disambiguate types to the same degree as C++. The only way to resolve this ambiguity is to use the %rename directive to rename one of the overloaded methods (effectively eliminating overloading).
- Typechecking may be partial. For example, if working with arrays, the typecheck code might simply check the type of the first array element and use that to dispatch to the correct function. Subsequent "in" typemaps would then perform more extensive type-checking.
- Make sure you read the section on overloading in the "[SWIG and C++](#)" chapter.

## 10.10 More about %apply and %clear

In order to implement certain kinds of program behavior, it is sometimes necessary to write sets of typemaps. For example, to support output arguments, one often writes a set of typemaps like this:

```
%typemap(in,numinputs=0) int *OUTPUT (int temp) {
    $1 = &temp;
}
%typemap(argout) int *OUTPUT {
    // return value somehow
}
```

To make it easier to apply the typemap to different argument types and names, the %apply directive performs a copy of all typemaps from one type to another. For example, if you specify this,

```
%apply int *OUTPUT { int *retvalue, int32 *output };
```

then all of the int \*OUTPUT typemaps are copied to int \*retvalue and int32 \*output.

However, there is a subtle aspect of %apply that needs more description. Namely, %apply does not overwrite a typemap rule if it is already defined for the target datatype. This behavior allows you to do two things:

- You can specialize parts of a complex typemap rule by first defining a few typemaps and then using %apply to incorporate the remaining pieces.
- Sets of different typemaps can be applied to the same datatype using repeated %apply directives.

For example:

```
%typemap(in) int *INPUT (int temp) {
    temp = ... get value from $input ...;
    $1 = &temp;
}

%typemap(check) int *POSITIVE {
    if (*$1 <= 0) {
        SWIG_exception(SWIG_ValueError,"Expected a positive number!\n");
        return NULL;
    }
}

...
%apply int *INPUT { int *invalue };
%apply int *POSITIVE { int *invalue };
```

Since %apply does not overwrite or replace any existing rules, the only way to reset behavior is to use the %clear directive. %clear removes all typemap rules defined for a specific datatype. For example:

```
%clear int *invalue;
```

## 10.11 Reducing wrapper code size

Since the code supplied to a typemap is inlined directly into wrapper functions, typemaps can result in a tremendous amount of code bloat. For example, consider this typemap for an array:

```
%typemap(in) float [ANY] {
    int i;
    if (!PySequence_Check($input)) {
        PyErr_SetString(PyExc_ValueError, "Expected a sequence");
        return NULL;
    }
    if (PySequence_Length($input) != $1_dim0) {
        PyErr_SetString(PyExc_ValueError, "Size mismatch. Expected $1_dim0 elements");
        return NULL;
    }
    $1 = (float) malloc($1_dim0*sizeof(float));
    for (i = 0; i < $1_dim0; i++) {
        PyObject *o = PySequence_GetItem($input, i);
        if (PyNumber_Check(o)) {
            $1[i] = (float) PyFloat_AsDouble(o);
        } else {
            PyErr_SetString(PyExc_ValueError, "Sequence elements must be numbers");
            return NULL;
        }
    }
}
```

If you had a large interface with hundreds of functions all accepting array parameters, this typemap would be replicated repeatedly—generating a huge amount of code. A better approach might be to consolidate some of the typemap into a function. For example:

```
%{
/* Define a helper function */
static float *
convert_float_array(PyObject *input, int size) {
    int i;
    float *result;
    if (!PySequence_Check(input)) {
        PyErr_SetString(PyExc_ValueError, "Expected a sequence");
        return NULL;
    }
    if (PySequence_Length(input) != size) {
        PyErr_SetString(PyExc_ValueError, "Size mismatch. ");
        return NULL;
    }
    result = (float) malloc(size*sizeof(float));
    for (i = 0; i < size; i++) {
        PyObject *o = PySequence_GetItem(input, i);
        if (PyNumber_Check(o)) {
            result[i] = (float) PyFloat_AsDouble(o);
        } else {
            PyErr_SetString(PyExc_ValueError, "Sequence elements must be numbers");
            free(result);
            return NULL;
        }
    }
    return result;
}
%}

%typemap(in) float [ANY] {
    $1 = convert_float_array($input, $1_dim0);
    if (!$1) return NULL;
}
%}
```

## 10.12 Passing data between typemaps

It is also important to note that the primary use of local variables is to create stack-allocated objects for temporary use inside a wrapper function (this is faster and less-prone to error than allocating data on the heap). In general, the variables are not intended to pass information between different types of typemaps. However, this can be done if you realize that local names have the argument number appended to them. For example, you could do this:

```
%typemap(in) int *(int temp) {
    temp = (int) PyInt_AsLong($input);
    $1 = &temp;
}

%typemap(argout) int * {
    PyObject *o = PyInt_FromLong(temp$argsnum);
    ...
}
```

In this case, the `$argsnum` variable is expanded into the argument number. Therefore, the code will reference the appropriate local such as `temp1` and `temp2`. It should be noted that there are plenty of opportunities to break the universe here and that accessing locals in this manner should probably be avoided. At the very least, you should make sure that the typemaps sharing information have exactly the same types and names.

## 10.13 Where to go for more information?

The best place to find out more information about writing typemaps is to look in the SWIG library. Most language modules define all of their default behavior using typemaps. These are found in files such as `python.swg`, `perl5.swg`, `tcl8.swg` and so forth. The `typemaps.i` file in the library also contains numerous examples. You should look at these files to get a feel for how to define typemaps of your own. Some of the language modules support additional typemaps and further information is available in the individual chapters for each target language.

# 11 Customization Features

- [Exception handling with %exception](#)
  - ◆ [Handling exceptions in C code](#)
  - ◆ [Exception handling with longjmp\(\)](#)
  - ◆ [Handling C++ exceptions](#)
  - ◆ [Defining different exception handlers](#)
  - ◆ [Using The SWIG exception library](#)
- [Object ownership and %newobject](#)
- [Features and the %feature directive](#)
  - ◆ [Features and default arguments](#)
  - ◆ [Feature example](#)

In many cases, it is desirable to change the default wrapping of particular declarations in an interface. For example, you might want to provide hooks for catching C++ exceptions, add assertions, or provide hints to the underlying code generator. This chapter describes some of these customization techniques. First, a discussion of exception handling is presented. Then, a more general-purpose customization mechanism known as "features" is described.

## 11.1 Exception handling with %exception

The `%exception` directive allows you to define a general purpose exception handler. For example, you can specify the following:

```
%exception {
    try {
        $action
    }
    catch (RangeError) {
        PyErr_SetString(PyExc_IndexError,"index out-of-bounds");
        return NULL;
    }
}
```

When defined, the code enclosed in braces is inserted directly into the low-level wrapper functions. The special symbol `$action` gets replaced with the actual operation to be performed (a function call, method invocation, attribute access, etc.). An exception handler remains in effect until it is explicitly deleted. This is done by using either `%exception` or `%noexception` with no code. For example:

```
%exception;    // Deletes any previously defined handler
```

**Compatibility note:** Previous versions of SWIG used a special directive `%except` for exception handling. That directive is still supported but is deprecated—`%exception` provides the same functionality, but is substantially more flexible.

### 11.1.1 Handling exceptions in C code

C has no formal exception handling mechanism so there are several approaches that might be used. A somewhat common technique is to simply set a special error code. For example:

```
/* File : except.c */

static char error_message[256];
static int error_status = 0;

void throw_exception(char *msg) {
    strncpy(error_message,msg,256);
    error_status = 1;
}
```



```

void clear_exception() {
    error_status = 0;
}
char *check_exception() {
    if (error_status) return error_message;
    else return NULL;
}

```

To use these functions, functions simply call `throw_exception()` to indicate an error occurred. For example :

```

double inv(double x) {
    if (x != 0) return 1.0/x;
    else {
        throw_exception("Division by zero");
        return 0;
    }
}

```

To catch the exception, you can write a simple exception handler such as the following (shown for Perl5) :

```

%exception {
    char *err;
    clear_exception();
    $action
    if ((err = check_exception())) {
        croak(err);
    }
}

```

In this case, when an error occurs, it is translated into a Perl error.

### 11.1.2 Exception handling with `longjmp()`

Exception handling can also be added to C code using the `<setjmp.h>` library. Here is a minimalistic implementation that relies on the C preprocessor :

```

/* File : except.c
   Just the declaration of a few global variables we're going to use */

#include <setjmp.h>
jmp_buf exception_buffer;
int exception_status;

/* File : except.h */
#include <setjmp.h>
extern jmp_buf exception_buffer;
extern int exception_status;

#define try if ((exception_status = setjmp(exception_buffer)) == 0)
#define catch(val) else if (exception_status == val)
#define throw(val) longjmp(exception_buffer, val)
#define finally else

/* Exception codes */

#define RangeError      1
#define DivisionByZero  2
#define OutOfMemory     3

```

Now, within a C program, you can do the following :

```
double inv(double x) {
    if (x) return 1.0/x;
    else throw(DivisionByZero);
}
```

Finally, to create a SWIG exception handler, write the following :

```
%{
#include "except.h"
}%

%exception {
    try {
        $action
    } catch(RangeError) {
        croak("Range Error");
    } catch(DivisionByZero) {
        croak("Division by zero");
    } catch(OutOfMemory) {
        croak("Out of memory");
    } finally {
        croak("Unknown exception");
    }
}
```

Note: This implementation is only intended to illustrate the general idea. To make it work better, you'll need to modify it to handle nested try declarations.

### 11.1.3 Handling C++ exceptions

Handling C++ exceptions is also straightforward. For example:

```
%exception {
    try {
        $action
    } catch(RangeError) {
        croak("Range Error");
    } catch(DivisionByZero) {
        croak("Division by zero");
    } catch(OutOfMemory) {
        croak("Out of memory");
    } catch(...) {
        croak("Unknown exception");
    }
}
```

The exception types need to be declared as classes elsewhere, possibly in a header file :

```
class RangeError {};
class DivisionByZero {};
class OutOfMemory {};
```

### 11.1.4 Defining different exception handlers

By default, the `%exception` directive creates an exception handler that is used for all wrapper functions that follow it. Unless there is a well-defined (and simple) error handling mechanism in place, defining one universal exception handler may be unwieldy and result in excessive code bloat since the handler is inlined into each wrapper function.

To fix this, you can be more selective about how you use the `%exception` directive. One approach is to only place it around critical pieces of code. For example:

```
%exception {
    ... your exception handler ...
}
/* Define critical operations that can throw exceptions here */

%exception;

/* Define non-critical operations that don't throw exceptions */
```

More precise control over exception handling can be obtained by attaching an exception handler to specific declaration name. For example:

```
%exception allocate {
    try {
        $action
    }
    catch (MemoryError) {
        croak("Out of memory");
    }
}
```

In this case, the exception handler is only attached to declarations named "allocate". This would include both global and member functions. The names supplied to %exception follow the same rules as for %rename described in the section on [Ambiguity resolution and renaming](#). For example, if you wanted to define an exception handler for a specific class, you might write this:

```
%exception Object::allocate {
    try {
        $action
    }
    catch (MemoryError) {
        croak("Out of memory");
    }
}
```

When a class prefix is supplied, the exception handler is applied to the corresponding declaration in the specified class as well as for identically named functions appearing in derived classes.

%exception can even be used to pinpoint a precise declaration when overloading is used. For example:

```
%exception Object::allocate(int) {
    try {
        $action
    }
    catch (MemoryError) {
        croak("Out of memory");
    }
}
```

Attaching exceptions to specific declarations is a good way to reduce code bloat. It can also be a useful way to attach exceptions to specific parts of a header file. For example:

```
%module example
%{
#include "someheader.h"
%}

// Define a few exception handlers for specific declarations
%exception Object::allocate(int) {
    try {
        $action
    }
    catch (MemoryError) {
        croak("Out of memory");
    }
}
```

```

}

%exception Object::getitem {
    try {
        $action
    }
    catch (RangeError) {
        croak("Index out of range");
    }
}

...
// Read a raw header file
#include "someheader.h"

```

**Compatibility note:** The `%exception` directive replaces the functionality provided by the deprecated "except" typemap. The typemap would allow exceptions to be thrown in the target language based on the return type of a function and was intended to be a mechanism for pinpointing specific declarations. However, it never really worked that well and the new `%exception` directive is much better.

### 11.1.5 Using The SWIG exception library

The `exception.i` library file provides support for creating language independent exceptions in your interfaces. To use it, simply put an `%include exception.i` in your interface file. This creates a function `SWIG_exception()` that can be used to raise common scripting language exceptions in a portable manner. For example :

```

// Language independent exception handler
#include exception.i

%exception {
    try {
        $action
    } catch(RangeError) {
        SWIG_exception(SWIG_ValueError, "Range Error");
    } catch(DivisionByZero) {
        SWIG_exception(SWIG_DivisionByZero, "Division by zero");
    } catch(OutOfMemory) {
        SWIG_exception(SWIG_MemoryError, "Out of memory");
    } catch(...) {
        SWIG_exception(SWIG_RuntimeError, "Unknown exception");
    }
}

```

As arguments, `SWIG_exception()` takes an error type code (an integer) and an error message string. The currently supported error types are :

```

SWIG_MemoryError
SWIG_IOError
SWIG_RuntimeError
SWIG_IndexError
SWIG_TypeError
SWIG_DivisionByZero
SWIG_OverflowError
SWIG_SyntaxError
SWIG_ValueError
SWIG_SystemError
SWIG_UnknownError

```

Since the `SWIG_exception()` function is defined at the C-level it can be used elsewhere in SWIG. This includes typemaps and helper functions.

## 11.2 Object ownership and %newobject

A common problem in some applications is managing proper ownership of objects. For example, consider a function like this:

```
Foo *blah() {
    Foo *f = new Foo();
    return f;
}
```

If you wrap the function `blah()`, SWIG has no idea that the return value is a newly allocated object. As a result, the resulting extension module may produce a memory leak (SWIG is conservative and will never delete objects unless it knows for certain that the returned object was newly created).

To fix this, you can provide an extra hint to the code generator using the `%newobject` directive. For example:

```
%newobject blah;
Foo *blah();
```

`%newobject` works exactly like `%rename` and `%exception`. In other words, you can attach it to class members and parameterized declarations as before. For example:

```
%newobject ::blah();           // Only applies to global blah
%newobject Object::blah(int,double); // Only blah(int,double) in Object
%newobject *::copy;           // Copy method in all classes
...
```

When `%newobject` is supplied, many language modules will arrange to take ownership of the return value. This allows the value to be automatically garbage-collected when it is no longer in use. However, this depends entirely on the target language (a language module may also choose to ignore the `%newobject` directive).

Closely related to `%newobject` is a special typemap. The "newfree" typemap can be used to deallocate a newly allocated return value. It is only available on methods for which `%newobject` has been applied and is commonly used to clean-up string results. For example:

```
%typemap(newfree) char * "free($1);";
...
%newobject strdup;
...
char *strdup(const char *s);
```

In this case, the result of the function is a string in the target language. Since this string is a copy of the original result, the data returned by `strdup()` is no longer needed. The "newfree" typemap in the example simply releases this memory.

**Compatibility note:** Previous versions of SWIG had a special `%new` directive. However, unlike `%newobject`, it only applied to the next declaration. For example:

```
%new char *strdup(const char *s);
```

For now this is still supported but is deprecated.

**How to shoot yourself in the foot:** The `%newobject` directive is not a declaration modifier like the old `%new` directive. Don't write code like this:

```
%newobject
char *strdup(const char *s);
```

The results might not be what you expect.

## 11.3 Features and the %feature directive

Both %exception and %newobject are examples of a more general purpose customization mechanism known as "features." A feature is simply a user-definable property that is attached to specific declarations in an interface file. Features are attached using the %feature directive. For example:

```
%feature("except") Object::allocate {
    try {
        $action
    }
    catch (MemoryError) {
        croak("Out of memory");
    }
}

%feature("new", "1") *::copy;
```

In fact, the %exception and %newobject directives are really nothing more than macros involving %feature:

```
#define %exception %feature("except")
#define %newobject %feature("new", "1")
```

The %feature directive follows the same name matching rules as the %rename directive (which is in fact just a special form of %feature). This means that features can be applied with pinpoint accuracy to specific declarations if needed.

When a feature is defined, it is given a name and a value. Most commonly, the value is supplied after the declaration name as shown for the "except" example above. However, if the feature is simple, a value might be supplied as an extra argument as shown for the "new" feature.

A feature stays in effect until it is explicitly disabled. A feature is disabled by supplying a %feature directive with no value. For example:

```
%feature("except") Object::allocate;    // Removes any previously defined feature
```

If no declaration name is given, a global feature is defined. This feature is then attached to *every* declaration that follows. This is how global exception handlers are defined. For example:

```
/* Define a global exception handler */
%feature("except") {
    try {
        $action
    }
    ...
}

... bunch of declarations ...

/* Disable the exception handler */
%feature("except");
```

The %feature directive can be used with different syntax. The following are all equivalent:

```
%feature("except") Object::method { $action };
%feature("except") Object::method %{ $action %};
%feature("except") Object::method " $action ";
%feature("except", "$action") Object::method;
```

The syntax in the first variation will generate the { } delimiters used whereas the other variations will not. The %feature directive also accepts XML style attributes in the same way that typemaps will. Any number of attributes can be specified. The following is the generic syntax for features:

```
%feature("name", "value", attribute1="AttributeValue1") symbol;
%feature("name", attribute1="AttributeValue1") symbol {value};
%feature("name", attribute1="AttributeValue1") symbol {%value%};
%feature("name", attribute1="AttributeValue1") symbol "value";
```

More than one attribute can be specified using a comma separated list. The Java module is an example that uses attributes in `%feature("except")`. The `throws` attribute specifies the name of a Java class to add to a proxy method's `throws` clause. In the following example, `MyExceptionClass` is the name of the Java class for adding to the `throws` clause.

```
%feature("except", throws="MyExceptionClass") Object::method {
    try {
        $action
    } catch (...) {
        ... code to throw a MyExceptionClass Java exception ...
    }
};
```

Further details can be obtained from the [Java exception handling](#) section.

### 11.3.1 Features and default arguments

SWIG treats methods with default arguments as separate overloaded methods as detailed in the [default arguments](#) section. Any `%feature` targeting a method with default arguments will apply to all the extra overloaded methods that SWIG generates if the default arguments are specified in the feature. If the default arguments are not specified in the feature, then the feature will match that exact wrapper method only and not the extra overloaded methods that SWIG generates. For example:

```
%feature("except") void hello(int i=0, double d=0.0);
void hello(int i=0, double d=0.0);
```

will apply the feature to all three wrapper methods, that is:

```
void hello(int i, double d);
void hello(int i);
void hello();
```

If the default arguments are not specified in the feature:

```
%feature("except") void hello(int i, double d);
void hello(int i=0, double d=0.0);
```

then the feature will only apply to this wrapper method:

```
void hello(int i, double d);
```

and not these wrapper methods:

```
void hello(int i);
void hello();
```

If [compactdefaultargs](#) are being used, then the difference between specifying or not specifying default arguments in a feature is not applicable as just one wrapper is generated.

**Compatibility note:** The different behaviour of features specified with or without default arguments was introduced in SWIG-1.3.23 when the approach to wrapping methods with default arguments was changed.

### 11.3.2 Feature example

As has been shown earlier, the intended use for the `%feature` directive is as a highly flexible customization mechanism that can be used to annotate declarations with additional information for use by specific target language modules. Another example is in

the Python module. You might use `%feature` to rewrite proxy/shadow class code as follows:

```
%module example
%rename(bar_id) bar(int,double);

// Rewrite bar() to allow some nice overloading

%feature("shadow") Foo::bar(int) %{
def bar(*args):
    if len(args) == 3:
        return apply(examplec.Foo_bar_id,args)
    return apply(examplec.Foo_bar,args)
%}

class Foo {
public:
    int bar(int x);
    int bar(int x, double y);
}
```

Further details of `%feature` usage is described in the documentation for specific language modules.



# 12 Contracts

- [The %contract directive](#)
- [%contract and classes](#)
- [Constant aggregation and %aggregate\\_check](#)
- [Notes](#)

A common problem that arises when wrapping C libraries is that of maintaining reliability and checking for errors. The fact of the matter is that many C programs are notorious for not providing error checks. Not only that, when you expose the internals of an application as a library, it often becomes possible to crash it simply by providing bad inputs or using it in a way that wasn't intended.

This chapter describes SWIG's support for software contracts. In the context of SWIG, a contract can be viewed as a runtime constraint that is attached to a declaration. For example, you can easily attach argument checking rules, check the output values of a function and more. When one of the rules is violated by a script, a runtime exception is generated rather than having the program continue to execute.

## 12.1 The %contract directive

Contracts are added to a declaration using the %contract directive. Here is a simple example:

```
%contract sqrt(double x) {
  require:
    x >= 0;
  ensure:
    sqrt >= 0;
}

...
double sqrt(double);
```

In this case, a contract is being added to the `sqrt ( )` function. The %contract directive must always appear before the declaration in question. Within the contract there are two sections, both of which are optional. The `require:` section specifies conditions that must hold before the function is called. Typically, this is used to check argument values. The `ensure:` section specifies conditions that must hold after the function is called. This is often used to check return values or the state of the program. In both cases, the conditions that must hold must be specified as boolean expressions.

In the above example, we're simply making sure that `sqrt()` returns a non-negative number (if it didn't, then it would be broken in some way).

Once a contract has been specified, it modifies the behavior of the resulting module. For example:

```
>>> example.sqrt(2)
1.4142135623730951
>>> example.sqrt(-2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: Contract violation: require: (arg1>=0)
>>>
```

## 12.2 %contract and classes

The %contract directive can also be applied to class methods and constructors. For example:

```
%contract Foo::bar(int x, int y) {
  require:
    x > 0;
  ensure:
```

```

    bar > 0;
}

%contract Foo::Foo(int a) {
require:
    a > 0;
}

class Foo {
public:
    Foo(int);
    int bar(int, int);
};

```

The way in which `%contract` is applied is exactly the same as the `%feature` directive. Thus, any contract that you specified for a base class will also be attached to inherited methods. For example:

```

class Spam : public Foo {
public:
    int bar(int,int);    // Gets contract defined for Foo::bar(int,int)
};

```

In addition to this, separate contracts can be applied to both the base class and a derived class. For example:

```

%contract Foo::bar(int x, int) {
require:
    x > 0;
}

%contract Spam::bar(int, int y) {
require:
    y > 0;
}

class Foo {
public:
    int bar(int,int);    // Gets Foo::bar contract.
};

class Spam : public Foo {
public:
    int bar(int,int);    // Gets Foo::bar and Spam::bar contract
};

```

When more than one contract is applied, the conditions specified in a "require:" section are combined together using a logical-AND operation. In other words conditions specified for the base class and conditions specified for the derived class all must hold. In the above example, this means that both the arguments to `Spam::bar` must be positive.

## 12.3 Constant aggregation and `%aggregate_check`

Consider an interface file that contains the following code:

```

#define UP      1
#define DOWN    2
#define RIGHT   3
#define LEFT    4

void move(SomeObject *, int direction, int distance);

```

One thing you might want to do is impose a constraint on the direction parameter to make sure it's one of a few accepted values. To do that, SWIG provides an easy to use macro `%aggregate_check()` that works like this:

```

%aggregate_check(int, check_direction, UP, DOWN, LEFT, RIGHT);

```

This merely defines a utility function of the form

```
int check_direction(int x);
```

That checks the argument `x` to see if it is one of the values listed. This utility function can be used in contracts. For example:

```
%aggregate_check(int, check_direction, UP, DOWN, RIGHT, LEFT);
```

```
%contract move(SomeObject *, int direction, in) {
require:
    check_direction(direction);
}
```

```
#define UP      1
#define DOWN    2
#define RIGHT   3
#define LEFT    4
```

```
void move(SomeObject *, int direction, int distance);
```

Alternatively, it can be used in typemaps and other directives. For example:

```
%aggregate_check(int, check_direction, UP, DOWN, RIGHT, LEFT);
```

```
%typemap(check) int direction {
    if (!check_direction($1)) SWIG_exception(SWIG_ValueError, "Bad direction");
}
```

```
#define UP      1
#define DOWN    2
#define RIGHT   3
#define LEFT    4
```

```
void move(SomeObject *, int direction, int distance);
```

Regrettably, there is no automatic way to perform similar checks with enums values. Maybe in a future release.

## 12.4 Notes

Contract support was implemented by Songyan (Tiger) Feng and first appeared in SWIG-1.3.20.

# 13 Variable Length Arguments

- [Introduction](#)
- [The Problem](#)
- [Default varargs support](#)
- [Argument replacement using %varargs](#)
- [Varargs and typemaps](#)
- [Varargs wrapping with libffi](#)
- [Wrapping of va\\_list](#)
- [C++ Issues](#)
- [Discussion](#)

(a.k.a, "The horror. The horror.")

This chapter describes the problem of wrapping functions that take a variable number of arguments. For instance, generating wrappers for the `C printf()` family of functions.

This topic is sufficiently advanced to merit its own chapter. In fact, support for varargs is an often requested feature that was first added in SWIG-1.3.12. Most other wrapper generation tools have wisely chosen to avoid this issue.

## 13.1 Introduction

Some C and C++ programs may include functions that accept a variable number of arguments. For example, most programmers are familiar with functions from the C library such as the following:

```
int printf(const char *fmt, ...)
int fprintf(FILE *, const char *fmt, ...);
int sprintf(char *s, const char *fmt, ...);
```

Although there is probably little practical purpose in wrapping these specific C library functions in a scripting language (what would be the point?), a library may include its own set of special functions based on a similar API. For example:

```
int traceprintf(const char *fmt, ...);
```

In this case, you may want to have some kind of access from the target language.

Before describing the SWIG implementation, it is important to discuss the common uses of varargs that you are likely to encounter in real programs. Obviously, there are the `printf()` style output functions as shown. Closely related to this would be `scanf()` style input functions that accept a format string and a list of pointers into which return values are placed. However, variable length arguments are also sometimes used to write functions that accept a NULL-terminated list of pointers. A good example of this would be a function like this:

```
int execlp(const char *path, const char *arg1, ...)
...

/* Example */
execlp("ls", "ls", "-l", NULL);
```

In addition, varargs is sometimes used to fake default arguments in older C libraries. For instance, the low level `open()` system call is often declared as a varargs function so that it will accept two or three arguments:

```
int open(const char *path, int oflag, ...);
...

/* Examples */
f = open("foo", O_RDONLY);
g = open("bar", O_WRONLY | O_CREAT, 0644);
```

Finally, to implement a varargs function, recall that you have to use the C library functions defined in `<stdarg.h>`. For example:

```
List make_list(const char *s, ...) {
    va_list ap;
    List *x = new List();
    ...
    va_start(ap, s);
    while (s) {
        x.append(s);
        s = va_arg(ap, const char *);
    }
    va_end(ap);
    return x;
}
```

## 13.2 The Problem

Generating wrappers for a variable length argument function presents a number of special challenges. Although C provides support for implementing functions that receive variable length arguments, there are no functions that can go in the other direction. Specifically, you can't write a function that dynamically creates a list of arguments and which invokes a varargs function on your behalf.

Although it is possible to write functions that accept the special type `va_list`, this is something entirely different. You can't take a `va_list` structure and pass it in place of the variable length arguments to another varargs function. It just doesn't work.

The reason this doesn't work has to do with the way that function calls get compiled. For example, suppose that your program has a function call like this:

```
printf("Hello %s. Your number is %d\n", name, num);
```

When the compiler looks at this, it knows that you are calling `printf()` with exactly three arguments. Furthermore, it knows that the number of arguments as well as their types and sizes is *never* going to change during program execution. Therefore, this gets turned to machine code that sets up a three-argument stack frame followed by a call to `printf()`.

In contrast, suppose you attempted to make some kind of wrapper around `printf()` using code like this:

```
int wrap_printf(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    ...
    printf(fmt, ap);
    ...
    va_end(ap);
};
```

Although this code might compile, it won't do what you expect. This is because the call to `printf()` is compiled as a procedure call involving only two arguments. However, clearly a two-argument configuration of the call stack is completely wrong if your intent is to pass an arbitrary number of arguments to the real `printf()`. Needless to say, it won't work.

Unfortunately, the situation just described is exactly the problem faced by wrapper generation tools. In general, the number of passed arguments will not be known until run-time. To make matters even worse, you won't know the types and sizes of arguments until run-time as well. Needless to say, there is no obvious way to make the C compiler generate code for a function call involving an unknown number of arguments of unknown types.

In theory, it *is* possible to write a wrapper that does the right thing. However, this involves knowing the underlying ABI for the target platform and language as well as writing special purpose code that manually constructed the call stack before making a procedure call. Unfortunately, both of these tasks require the use of inline assembly code. Clearly, that's the kind of solution you would much rather avoid.

With this nastiness in mind, SWIG provides a number of solutions to the varargs wrapping problem. Most of these solutions are compromises that provide limited varargs support without having to resort to assembly language. However, SWIG can also support real varargs wrapping (with stack-frame manipulation) if you are willing to get hands dirty. Keep reading.

## 13.3 Default varargs support

When variable length arguments appear in an interface, the default behavior is to drop the variable argument list entirely, replacing them with a single NULL pointer. For example, if you had this function,

```
void traceprintf(const char *fmt, ...);
```

it would be wrapped as if it had been declared as follows:

```
void traceprintf(const char *fmt);
```

When the function is called inside the wrappers, it is called as follows:

```
traceprintf(arg1, NULL);
```

Arguably, this approach seems to defeat the whole point of variable length arguments. However, this actually provides enough support for many simple kinds of varargs functions to still be useful. For instance, you could make function calls like this (in Python):

```
>>> traceprintf("Hello World")
>>> traceprintf("Hello %s. Your number is %d\n" % (name, num))
```

Notice how string formatting is being done in Python instead of C.

## 13.4 Argument replacement using %varargs

Instead of dropping the variable length arguments, an alternative approach is to replace ( . . . ) with a set of suitable arguments. SWIG provides a special %varargs directive that can be used to do this. For example,

```
%varargs(int mode = 0) open;
...
int open(const char *path, int oflags, ...);
```

is equivalent to this:

```
int open(const char *path, int oflags, int mode = 0);
```

In this case, %varargs is simply providing more specific information about the extra arguments that might be passed to a function. If the parameters to a varargs function are of uniform type, %varargs can also accept a numerical argument count as follows:

```
%varargs(10,char *arg = NULL) execlp;
...
int execlp(const char *path, const char *arg1, ...);
```

This would wrap execlp( ) as a function that accepted up to 10 optional arguments. Depending on the application, this may be more than enough for practical purposes.

Argument replacement is most appropriate in cases where the types of the extra arguments is uniform and the maximum number of arguments is known. When replicated argument replacement is used, at least one extra argument is added to the end of the arguments when making the function call. This argument serves as a sentinel to make sure the list is properly terminated. It has the same value as that supplied to the %varargs directive.

Argument replacement is not as useful when working with functions that accept mixed argument types such as `printf()`. Providing general purpose wrappers to such functions presents special problems (covered shortly).

## 13.5 Varargs and typemaps

Variable length arguments may be used in typemap specifications. For example:

```
%typemap(in) (...) {
    // Get variable length arguments (somehow)
    ...
}

%typemap(in) (const char *fmt, ...) {
    // Multi-argument typemap
}
```

However, this immediately raises the question of what "type" is actually used to represent `(...)`. For lack of a better alternative, the type of `(...)` is set to `void *`. Since there is no way to dynamically pass arguments to a varargs function (as previously described), the `void *` argument value is intended to serve as a place holder for storing some kind of information about the extra arguments (if any). In addition, the default behavior of SWIG is to pass the `void *` value as an argument to the function. Therefore, you could use the pointer to hold a valid argument value if you wanted.

To illustrate, here is a safer version of wrapping `printf()` in Python:

```
%typemap(in) (const char *fmt, ...) {
    $1 = "%s";
    $2 = (void *) PyString_AsString($input); /* Fix format string to %s */
};
...
int printf(const char *fmt, ...);
```

In this example, the format string is implicitly set to `"%s"`. This prevents a program from passing a bogus format string to the extension. Then, the passed input object is decoded and placed in the `void *` argument defined for the `(...)` argument. When the actual function call is made, the underlying wrapper code will look roughly like this:

```
wrap_printf() {
    char *arg1;
    void *arg2;
    int result;

    arg1 = "%s";
    arg2 = (void *) PyString_AsString(arg2obj);
    ...
    result = printf(arg1, arg2);
    ...
}
```

Notice how both arguments are passed to the function and it does what you would expect.

The next example illustrates a more advanced kind of varargs typemap. Disclaimer: this requires special support in the target language module and is not guaranteed to work with all SWIG modules at this time. It also starts to illustrate some of the more fundamental problems with supporting varargs in more generality.

If a typemap is defined for any form of `(...)`, many SWIG modules will generate wrappers that accept a variable number of arguments as input and will make these arguments available in some form. The precise details of this depends on the language module being used (consult the appropriate chapter for more details). However, suppose that you wanted to create a Python wrapper for the `execvp()` function shown earlier. To do this using a typemap instead of using `%varargs`, you might first write a typemap like this:

```
%typemap(in) (...)(char *args[10]) {
```

```

int i;
int argc;
for (i = 0; i < 10; i++) args[i] = 0;
argc = PyTuple_Size(varargs);
if (argc > 10) {
    PyErr_SetString(PyExc_ValueError, "Too many arguments");
    return NULL;
}
for (i = 0; i < argc; i++) {
    PyObject *o = PyTuple_GetItem(varargs, i);
    if (!PyString_Check(o)) {
        PyErr_SetString(PyExc_ValueError, "Expected a string");
        return NULL;
    }
    args[i] = PyString_AsString(o);
}
$1 = (void *) args;
}

```

In this typemap, the special variable `varargs` is a tuple holding all of the extra arguments passed (this is specific to the Python module). The typemap then pulls this apart and sticks the values into the array of strings `args`. Then, the array is assigned to `$1` (recall that this is the `void *` variable corresponding to `(...)`). However, this assignment is only half of the picture-----clearly this alone is not enough to make the function work. To patch everything up, you have to rewrite the underlying action code using the `%feature` directive like this:

```

%feature("action") execlp {
    char *args = (char **) arg3;
    result = execlp(arg1, arg2, args[0], args[1], args[2], args[3], args[4],
                    args[5], args[6], args[7], args[8], args[9], NULL);
}

int execlp(const char *path, const char *arg, ...);

```

This patches everything up and creates a function that more or less works. However, don't try explaining this to your coworkers unless you know for certain that they've had several cups of coffee. If you really want to elevate your guru status and increase your job security, continue to the next section.

## 13.6 Varargs wrapping with libffi

All of the previous examples have relied on features of SWIG that are portable and which don't rely upon any low-level machine-level details. In many ways, they have all dodged the real issue of variable length arguments by recasting a `varargs` function into some weaker variation with a fixed number of arguments of known types. In many cases, this works perfectly fine. However, if you want more generality than this, you need to bring out some bigger guns.

One way to do this is to use a special purpose library such as `libffi` (<http://sources.redhat.com/libffi>). `libffi` is a library that allows you to dynamically construct call-stacks and invoke procedures in a relatively platform independent manner. Details about the library can be found in the `libffi` distribution and are not repeated here.

To illustrate the use of `libffi`, suppose that you *really* wanted to create a wrapper for `execlp()` that accepted *any* number of arguments. To do this, you might make a few adjustments to the previous example. For example:

```

/* Take an arbitrary number of extra arguments and place into an array
   of strings */

%typemap(in) (...) {
    char **argv;
    int    argc;
    int    i;

    argc = PyTuple_Size(varargs);
    argv = (char **) malloc(sizeof(char *)*(argc+1));
    for (i = 0; i < argc; i++) {

```



```

PyObject *o = PyTuple_GetItem(varargs,i);
if (!PyString_Check(o)) {
    PyErr_SetString(PyExc_ValueError,"Expected a string");
    free(argv);
    return NULL;
}
argv[i] = PyString_AsString(o);
}
argv[i] = NULL;
$1 = (void *) argv;
}

/* Rewrite the function call, using libffi */

%feature("action") execlp {
    int      i, vc;
    ffi_cif  cif;
    ffi_type **types;
    void     **values;
    char     **args;

    vc = PyTuple_Size(varargs);
    types = (ffi_type **) malloc((vc+3)*sizeof(ffi_type *));
    values = (void **) malloc((vc+3)*sizeof(void *));
    args = (char **) arg3;

    /* Set up path parameter */
    types[0] = &ffi_type_pointer;
    values[0] = &arg1;

    /* Set up first argument */
    types[1] = &ffi_type_pointer;
    values[1] = &arg2;

    /* Set up rest of parameters */
    for (i = 0; i <= vc; i++) {
        types[2+i] = &ffi_type_pointer;
        values[2+i] = &args[i];
    }
    if (ffi_prep_cif(&cif, FFI_DEFAULT_ABI, vc+3,
                    &ffi_type_uint, types) == FFI_OK) {
        ffi_call(&cif, (void (*)()) execlp, &result, values);
    } else {
        PyErr_SetString(PyExc_RuntimeError, "Whoa!!!!");
        free(types);
        free(values);
        free(arg3);
        return NULL;
    }
    free(types);
    free(values);
    free(arg3);
}

/* Declare the function. Whew! */
int execlp(const char *path, const char *arg1, ...);

```

Looking at this example, you may start to wonder if SWIG is making life any easier. Given the amount of code involved, you might also wonder why you didn't just write a hand-crafted wrapper! Either that or you're wondering "why in the hell am I trying to wrap this varargs function in the first place?!?" Obviously, those are questions you'll have to answer for yourself.

As a more extreme example of libffi, here is some code that attempts to wrap `printf()`,

```

/* A wrapper for printf() using libffi */

%{

```

```

/* Structure for holding passed arguments after conversion */
typedef struct {
    int type;
    union {
        int    ivalue;
        double dvalue;
        void   *pvalue;
    } val;
} vtype;
enum { VT_INT, VT_DOUBLE, VT_POINTER };
%}

%typemap(in) (const char *fmt, ...) {
    vtype *argv;
    int    argc;
    int    i;

    /* Format string */
    $1 = PyString_AsString($input);

    /* Variable length arguments */
    argc = PyTuple_Size(varargs);
    argv = (vtype *) malloc(argc*sizeof(vtype));
    for (i = 0; i < argc; i++) {
        PyObject *o = PyTuple_GetItem(varargs,i);
        if (PyInt_Check(o)) {
            argv[i].type = VT_INT;
            argv[i].val.ivalue = PyInt_AsLong(o);
        } else if (PyFloat_Check(o)) {
            argv[i].type = VT_DOUBLE;
            argv[i].val.dvalue = PyFloat_AsDouble(o);
        } else if (PyString_Check(o)) {
            argv[i].type = VT_POINTER;
            argv[i].val.pvalue = (void *) PyString_AsString(o);
        } else {
            PyErr_SetString(PyExc_ValueError,"Unsupported argument type");
            free(argv);
            return NULL;
        }
    }
    $2 = (void *) argv;
}

/* Rewrite the function call using libffi */
%feature("action") printf {
    int    i, vc;
    ffi_cif cif;
    ffi_type **types;
    void    **values;
    vtype    *args;

    vc = PyTuple_Size(varargs);
    types = (ffi_type **) malloc((vc+1)*sizeof(ffi_type *));
    values = (void **) malloc((vc+1)*sizeof(void *));
    args = (vtype *) arg2;

    /* Set up fmt parameter */
    types[0] = &ffi_type_pointer;
    values[0] = &arg1;

    /* Set up rest of parameters */
    for (i = 0; i < vc; i++) {
        switch(args[i].type) {
            case VT_INT:
                types[1+i] = &ffi_type_uint;
                values[1+i] = &args[i].val.ivalue;
                break;
            case VT_DOUBLE:

```

```

        types[l+i] = &ffi_type_double;
        values[l+i] = &args[i].val.dvalue;
        break;
    case VT_POINTER:
        types[l+i] = &ffi_type_pointer;
        values[l+i] = &args[i].val.pvalue;
        break;
    default:
        abort(); /* Whoa! We're seriously hosed */
        break;
    }
}
if (ffi_prep_cif(&cif, FFI_DEFAULT_ABI, vc+1,
                &ffi_type_uint, types) == FFI_OK) {
    ffi_call(&cif, (void (*)()) printf, &result, values);
} else {
    PyErr_SetString(PyExc_RuntimeError, "Whoa!!!!");
    free(types);
    free(values);
    free(args);
    return NULL;
}
free(types);
free(values);
free(args);
}

/* The function */
int printf(const char *fmt, ...);

```

Much to your amazement, it even seems to work if you try it:

```

>>> import example
>>> example.printf("Grade: %s    %d/60 = %0.2f%%\n", "Dave", 47, 47.0*100/60)
Grade: Dave    47/60 = 78.33%
>>>

```

Of course, there are still some limitations to consider:

```

>>> example.printf("la de da de da %s", 42)
Segmentation fault (core dumped)

```

And, on this note, we leave further exploration of libffi to the reader as an exercise. Although Python has been used as an example, most of the techniques in this section can be extrapolated to other language modules with a bit of work. The only details you need to know is how the extra arguments are accessed in each target language. For example, in the Python module, we used the special `varargs` variable to get these arguments. Modules such as Tcl8 and Perl5 simply provide an argument number for the first extra argument. This can be used to index into an array of passed arguments to get values. Please consult the chapter on each language module for more details.

## 13.7 Wrapping of `va_list`

Closely related to variable length argument wrapping, you may encounter functions that accept a parameter of type `va_list`. For example:

```
int vfprintf(FILE *f, const char *fmt, va_list ap);
```

As far as we know, there is no obvious way to wrap these functions with SWIG. This is because there is no documented way to assemble the proper `va_list` structure (there are no C library functions to do it and the contents of `va_list` are opaque). Not only that, the contents of a `va_list` structure are closely tied to the underlying call-stack. It's not clear that exporting a `va_list` would have any use or that it would work at all.

## 13.8 C++ Issues

Wrapping of C++ member functions that accept a variable number of arguments presents a number of challenges. By far, the easiest way to handle this is to use the `%varargs` directive. This is portable and it fully supports classes much like the `%rename` directive. For example:

```
%varargs (10, char * = NULL) Foo::bar;

class Foo {
public:
    virtual void bar(char *arg, ...);    // gets varargs above
};

class Spam: public Foo {
public:
    virtual void bar(char *arg, ...);    // gets varargs above
};
```

`%varargs` also works with constructors, operators, and any other C++ programming construct that accepts variable arguments.

Doing anything more advanced than this is likely to involve a serious world of pain. In order to use a library like `libffi`, you will need to know the underlying calling conventions and details of the C++ ABI. For instance, the details of how `this` is passed to member functions as well as any hidden arguments that might be used to pass additional information. These details are implementation specific and may differ between compilers and even different versions of the same compiler. Also, be aware that invoking a member function is further complicated if it is a virtual method. In this case, invocation might require a table lookup to obtain the proper function address (although you might be able to obtain an address by casting a bound pointer to a pointer to function as described in the C++ ARM section 18.3.4).

If you do decide to change the underlying action code, be aware that SWIG always places the `this` pointer in `arg1`. Other arguments are placed in `arg2`, `arg3`, and so forth. For example:

```
%feature("action") Foo::bar {
    ...
    result = arg1->bar(arg2, arg3, etc.);
    ...
}
```

Given the potential to shoot yourself in the foot, it is probably easier to reconsider your design or to provide an alternative interface using a helper function than it is to create a fully general wrapper to a `varargs` C++ member function.

## 13.9 Discussion

This chapter has provided a number of techniques that can be used to address the problem of variable length argument wrapping. If you care about portability and ease of use, the `%varargs` directive is probably the easiest way to tackle the problem. However, using `typemaps`, it is possible to do some very advanced kinds of wrapping.

One point of discussion concerns the structure of the `libffi` examples in the previous section. Looking at that code, it is not at all clear that this is the easiest way to solve the problem. However, there are a number of subtle aspects of the solution to consider—mostly concerning the way in which the problem has been decomposed. First, the example is structured in a way that tries to maintain separation between wrapper-specific information and the declaration of the function itself. The idea here is that you might structure your interface like this:

```
%typemap(const char *fmt, ...) {
    ...
}
%feature("action") traceprintf {
    ...
}
```

```
/* Include some header file with traceprintf in it */
#include "someheader.h"
```

Second, careful scrutiny will reveal that the typemaps involving ( . . . ) have nothing whatsoever to do with the libffi library. In fact, they are generic with respect to the way in which the function is actually called. This decoupling means that it will be much easier to consider other library alternatives for making the function call. For instance, if libffi wasn't supported on a certain platform, you might be able to use something else instead. You could use conditional compilation to control this:

```
#ifdef USE_LIBFFI
%feature("action") printf {
    ...
}
#endif
#ifdef USE_OTHERFFI
%feature("action") printf {
    ...
}
#endif
```

Finally, even though you might be inclined to just write a hand-written wrapper for varargs functions, the techniques used in the previous section have the advantage of being compatible with all other features of SWIG such as exception handling.

As a final word, some C programmers seem to have the assumption that the wrapping of variable length argument functions is an easily solved problem. However, this section has hopefully dispelled some of these myths. All things being equal, you are better off avoiding variable length arguments if you can. If you can't avoid them, please consider some of the simple solutions first. If you can't live with a simple solution, proceed with caution. At the very least, make sure you carefully read the section "A7.3.2 Function Calls" in Kernighan and Ritchie and make sure you fully understand the parameter passing conventions used for varargs. Also, be aware of the platform dependencies and reliability issues that this will introduce. Good luck.

# 14 Warning Messages

- [Introduction](#)
- [Warning message suppression](#)
- [Enabling additional warnings](#)
- [Issuing a warning message](#)
- [Commentary](#)
- [Warnings as errors](#)
- [Message output format](#)
- [Warning number reference](#)
  - ◆ [Deprecated features \(100–199\)](#)
  - ◆ [Preprocessor \(200–299\)](#)
  - ◆ [C/C++ Parser \(300–399\)](#)
  - ◆ [Types and typemaps \(400–499\)](#)
  - ◆ [Code generation \(500–599\)](#)
  - ◆ [Language module specific \(800–899\)](#)
  - ◆ [User defined \(900–999\)](#)
- [History](#)

## 14.1 Introduction

During compilation, SWIG may generate a variety of warning messages. For example:

```
example.i:16: Warning(501): Overloaded declaration ignored.  bar(double)
example.i:15: Warning(501): Previous declaration is bar(int)
```

Typically, warning messages indicate non-fatal problems with the input where the generated wrapper code will probably compile, but it may not work like you expect.

## 14.2 Warning message suppression

All warning messages have a numeric code that is shown in the warning message itself. To suppress the printing of a warning message, a number of techniques can be used. First, you can run SWIG with the `-w` command line option. For example:

```
% swig -python -w501 example.i
% swig -python -w501,505,401 example.i
```

Alternatively, warnings can be suppressed by inserting a special preprocessor pragma into the input file:

```
%module example
#pragma SWIG nowarn=501
#pragma SWIG nowarn=501,505,401
```

Finally, code-generation warnings can be disabled on a declaration by declaration basis using the `%warnfilter` directive. For example:

```
%module example
%warnfilter(501) foo;
...
int foo(int);
int foo(double);           // Silently ignored.
```

The `%warnfilter` directive has the same semantics as other declaration modifiers like `%rename`, `%ignore`, and `%feature`. For example, if you wanted to suppress a warning for a method in a class hierarchy, you could do this:

```
%warnfilter(501) Object::foo;
class Object {
```

```

public:
    int foo(int);
    int foo(double);    // Silently ignored
    ...
};

class Derived : public Object {
public:
    int foo(int);
    int foo(double);    // Silently ignored
    ...
};

```

Warnings can be suppressed for an entire class by supplying a class name. For example:

```

%warnfilter(501) Object;

class Object {
public:
    ...                // All 501 warnings ignored in class
};

```

There is no option to suppress all SWIG warning messages. The warning messages are there for a reason—to tell you that something may be *broken* in your interface. Ignore the warning messages at your own peril.

## 14.3 Enabling additional warnings

Some warning messages are disabled by default and are generated only to provide additional diagnostics. All warning messages can be enabled using the `-Wall` option. For example:

```
% swig -Wall -python example.i
```

When `-Wall` is used, all other warning filters are disabled.

To selectively turn on extra warning messages, you can use the directives and options in the previous section—simply add a "+" to all warning numbers. For example:

```
% swig -w+309,+452 example.i
```

or

```
#pragma SWIG nowarn=+309,+452
```

or

```
%warnfilter(+309,+452) foo;
```

Note: selective enabling of warnings with `%warnfilter` overrides any global settings you might have made using `-w` or `#pragma`.

## 14.4 Issuing a warning message

Warning messages can be issued from an interface file using a number of directives. The `%warn` directive is the most simple:

```
%warn "750:This is your last warning!"
```

All warning messages are optionally prefixed by the warning number to use. If you are generating your own warnings, make sure you don't use numbers defined in the table at the end of this section.

The `%ignorewarn` directive is the same as `%ignore` except that it issues a warning message whenever a matching declaration is found. For example:

```
%ignorewarn("362:operator= ignored") operator=;
```

Warning messages can be associated with typemaps using the `warning` attribute of a typemap declaration. For example:

```
%typemap(in, warning="751:You are really going to regret this") blah * {
    ...
}
```

In this case, the warning message will be printed whenever the typemap is actually used.

## 14.5 Commentary

The ability to suppress warning messages is really only provided for advanced users and is not recommended in normal use. There are no plans to provide symbolic names or options that identify specific types or groups of warning messages—the numbers must be used explicitly.

Certain types of SWIG problems are errors. These usually arise due to parsing errors (bad syntax) or semantic problems for which there is no obvious recovery. There is no mechanism for suppressing error messages.

## 14.6 Warnings as errors

Warnings can be handled as errors by using the `-Werror` command line option. This will cause SWIG to exit with a non successful exit code if a warning is encountered.

## 14.7 Message output format

The output format for both warnings and errors can be selected for integration with your favourite IDE/editor. Editors and IDEs can usually parse error messages and if in the appropriate format will easily take you directly to the source of the error. The standard format is used by default except on Windows where the Microsoft format is used by default. These can be overridden using command line options, for example:

```
$ swig -python -Fstandard example.i
example.i:4: Syntax error in input.
$ swig -python -Fmicrosoft example.i
example.i(4): Syntax error in input.
```

## 14.8 Warning number reference

### 14.8.1 Deprecated features (100–199)

- 101. Deprecated `%extern` directive.
- 102. Deprecated `%val` directive.
- 103. Deprecated `%out` directive.
- 104. Deprecated `%disabledoc` directive.
- 105. Deprecated `%enabledoc` directive.
- 106. Deprecated `%doonly` directive.
- 107. Deprecated `%style` directive.
- 108. Deprecated `%localstyle` directive.
- 109. Deprecated `%title` directive.
- 110. Deprecated `%section` directive.
- 111. Deprecated `%subsection` directive.
- 112. Deprecated `%subsubsection` directive.
- 113. Deprecated `%addmethods` directive.



- 114. Deprecated `%readonly` directive.
- 115. Deprecated `%readwrite` directive.
- 116. Deprecated `%except` directive.
- 117. Deprecated `%new` directive.
- 118. Deprecated `%typemap(except)`.
- 119. Deprecated `%typemap(ignore)`.
- 120. Deprecated command line option `(-c)`.

## 14.8.2 Preprocessor (200–299)

- 201. Unable to find 'filename'.
- 202. Could not evaluate 'expr'.

## 14.8.3 C/C++ Parser (300–399)

- 301. `class` keyword used, but not in C++ mode.
- 302. Identifier '*name*' redefined (ignored).
- 303. `%extend` defined for an undeclared class '*name*'.
- 304. Unsupported constant value (ignored).
- 305. Bad constant value (ignored).
- 306. '*identifier*' is private in this context.
- 307. Can't set default argument value (ignored)
- 308. Namespace alias '*name*' not allowed here. Assuming '*name*'
- 309. `[private | protected]` inheritance ignored.
- 310. Template '*name*' was already wrapped as '*name*' (ignored)
- 311. Template partial specialization not supported.
- 312. Nested classes not currently supported (ignored).
- 313. Unrecognized extern type "*name*" (ignored).
- 314. '*identifier*' is a *lang* keyword.
- 315. Nothing known about '*identifier*'.
- 316. Repeated `%module` directive.
- 317. Specialization of non-template '*name*'.
- 318. Instantiation of template *name* is ambiguous. Using *templ* at *file:line*
- 319. No access specifier given for base class *name* (ignored).
- 320. Explicit template instantiation ignored.
- 321. *identifier* conflicts with a built-in name.
- 322. Redundant redeclaration of '*name*'.
- 350. operator `new` ignored.
- 351. operator `delete` ignored.
- 352. operator `+` ignored.
- 353. operator `-` ignored.
- 354. operator `*` ignored.
- 355. operator `/` ignored.
- 356. operator `%` ignored.
- 357. operator `^` ignored.
- 358. operator `&` ignored.
- 359. operator `|` ignored.
- 360. operator `~` ignored.
- 361. operator `!` ignored.
- 362. operator `=` ignored.
- 363. operator `<` ignored.
- 364. operator `>` ignored.
- 365. operator `+=` ignored.
- 366. operator `-=` ignored.
- 367. operator `*=` ignored.
- 368. operator `/=` ignored.

- 369. operator%= ignored.
- 370. operator^= ignored.
- 371. operator&= ignored.
- 372. operator|= ignored.
- 373. operator<< ignored.
- 374. operator>> ignored.
- 375. operator<<= ignored.
- 376. operator>>= ignored.
- 377. operator== ignored.
- 378. operator!= ignored.
- 379. operator<= ignored.
- 380. operator>= ignored.
- 381. operator&& ignored.
- 382. operator|| ignored.
- 383. operator++ ignored.
- 384. operator-- ignored.
- 385. operator, ignored.
- 386. operator-<\* ignored.
- 387. operator-< ignored.
- 388. operator() ignored.
- 389. operator[] ignored.
- 390. operator+ ignored (unary).
- 391. operator- ignored (unary).
- 392. operator\* ignored (unary).
- 393. operator& ignored (unary).
- 394. operator new[] ignored.
- 395. operator delete[] ignored.

#### 14.8.4 Types and typemaps (400–499)

- 401. Nothing known about class 'name'. Ignored.
- 402. Base class 'name' is incomplete.
- 403. Class 'name' might be abstract.
- 450. Deprecated typemap feature (\$source/\$target).
- 451. Setting const char \* variable may leak memory.
- 452. Reserved
- 453. Can't apply (pattern). No typemaps are defined.
- 460. Unable to use type *type* as a function argument.
- 461. Unable to use return type *type* in function *name*.
- 462. Unable to set variable of type *type*.
- 463. Unable to read variable of type *type*.
- 464. Unsupported constant value.
- 465. Unable to handle type *type*.
- 466. Unsupported variable type *type*.
- 467. Overloaded *declaration* not supported (no type checking rule for 'type')
- 468. No 'throw' typemap defined for exception type 'type'.

#### 14.8.5 Code generation (500–599)

- 501. Overloaded declaration ignored. *decl*
- 502. Overloaded constructor ignored. *decl*
- 503. Can't wrap '*identifier*' unless renamed to a valid identifier.
- 504. Function *name* must have a return type.
- 505. Variable length arguments discarded.
- 506. Can't wrap varargs with keyword arguments enabled.
- 507. Adding native function *name* not supported (ignored).

- 508. Declaration of '*name*' shadows declaration accessible via operator->() at *file:line*.
- 509. Overloaded *declaration* is shadowed by *declaration* at *file:line*.
- 510. Friend function '*name*' ignored.
- 511. Can't use keyword arguments with overloaded functions.
- 512. Overloaded *declaration* const ignored. Non-const method at *file:line* used.
- 513. Can't generate wrappers for unnamed struct/class.
- 514.
- 515.
- 516. Overloaded method *declaration* ignored. Method *declaration* at *file:line* used.

### 14.8.6 Language module specific (800–899)

- 801. Wrong name (corrected to '*name*'). (Ruby).
- 810. No jni typemap defined for *type* (Java).
- 811. No jtype typemap defined for *type* (Java).
- 812. No jstype typemap defined for *type* (Java).
- 813. Warning for *classname*: Base *baseclass* ignored. Multiple inheritance is not supported in Java. (Java).
- 814.
- 815. No javafinalize typemap defined for *type* (Java).
- 816. No javabody typemap defined for *type* (Java).
- 817. No javaout typemap defined for *type* (Java).
- 818. No javain typemap defined for *type* (Java).
- 819. No javadirectorin typemap defined for *type* (Java).
- 820. No javadirectorout typemap defined for *type* (Java).
- 821.
- 822. Covariant return types not supported in Java. Proxy method will return *basetype* (Java).
- 830. No ctype typemap defined for *type* (C#).
- 831. No cstype typemap defined for *type* (C#).
- 832. No cswtype typemap defined for *type* (C#).
- 833. Warning for *classname*: Base *baseclass* ignored. Multiple inheritance is not supported in C#. (C#).
- 834.
- 835. No csfinalize typemap defined for *type* (C#).
- 836. No csbody typemap defined for *type* (C#).
- 837. No csout typemap defined for *type* (C#).
- 838. No csin typemap defined for *type* (C#).
- 839.
- 840.
- 841.
- 842. Covariant return types not supported in C#. Proxy method will return *basetype* (C#).

### 14.8.7 User defined (900–999)

These numbers can be used by your own application.

## 14.9 History

The ability to control warning messages was first added to SWIG-1.3.12.

# 15 Working with Modules

- [The SWIG runtime code](#)
- [A word of caution about static libraries](#)
- [References](#)
- [Reducing the wrapper file size](#)

When first working with SWIG, users commonly start by creating a single module. That is, you might define a single SWIG interface that wraps some set of C/C++ code. You then compile all of the generated wrapper code into a module and use it. For large applications, however, this approach is problematic—the size of the generated wrapper code can be rather large. Moreover, it is probably easier to manage the target language interface when it is broken up into smaller pieces.

This chapter describes the problem of using SWIG in programs where you want to create a collection of modules.

## 15.1 The SWIG runtime code

Many of SWIG's target languages generate a set of functions commonly known as the "SWIG runtime." These functions are primarily related to the runtime type system which checks pointer types and performs other tasks such as proper casting of pointer values in C++. As a general rule, the statically typed target languages, such as Java, use the language's built in static type checking and have no need for a SWIG runtime. All the dynamically typed / interpreted languages rely on the SWIG runtime.

The runtime functions are private to each SWIG-generated module. That is, the runtime functions are declared with "static" linkage and are visible only to the wrapper functions defined in that module. The only problem with this approach is that when more than one SWIG module is used in the same application, those modules often need to share type information. This is especially true for C++ programs where SWIG must collect and share information about inheritance relationships that cross module boundaries.

To solve the problem of sharing information across modules, a pointer to the type information is stored in a global variable in the target language namespace. During module initialization, type information is loaded into the global data structure of type information from all modules.

This can present a problem with threads. If two modules try and load at the same time, the type information can become corrupt. SWIG currently does not provide any locking, and if you use threads, you must make sure that modules are loaded serially. Be careful if you use threads and the automatic module loading that some scripting languages provide. One solution is to load all modules before spawning any threads.

## 15.2 A word of caution about static libraries

When working with multiple SWIG modules, you should take care not to use static libraries. For example, if you have a static library `libfoo.a` and you link a collection of SWIG modules with that library, each module will get its own private copy of the library code inserted into it. This is very often **NOT** what you want and it can lead to unexpected or bizarre program behavior. When working with dynamically loadable modules, you should try to work exclusively with shared libraries.

## 15.3 References

Due to the complexity of working with shared libraries and multiple modules, it might be a good idea to consult an outside reference. John Levine's "Linkers and Loaders" is highly recommended.

## 15.4 Reducing the wrapper file size

Using multiple modules with the `%import` directive is the most common approach to modularising large projects. In this way a number of different wrapper files can be generated, thereby avoiding the generation of a single large wrapper file. There are a couple of alternative solutions for reducing the size of a wrapper file through the use of command line options and features.

**-fcompact**

This command line option will compact the size of the wrapper file without changing the code generated into the wrapper file. It simply removes blank lines and joins lines of code together. This is useful for compilers that have a maximum file size that can be handled.

**-fvirtual**

This command line option will remove the generation of superfluous virtual method wrappers. Consider the following inheritance hierarchy:

```
struct Base {
    virtual void method();
    ...
};

struct Derived : Base {
    virtual void method();
    ...
};
```

Normally wrappers are generated for both methods, whereas this command line option will suppress the generation of a wrapper for `Derived::method`. Normal polymorphic behaviour remains as `Derived::method` will still be called should you have a `Derived` instance and call the wrapper for `Base::method`.

**%feature("compactdefaultargs")**

This feature can reduce the number of wrapper methods when wrapping methods with default arguments. The section on [default arguments](#) discusses the feature and its limitations.

## 16 SWIG and C#

The purpose of the C# module is to offer an automated way of accessing existing C/C++ code from .NET languages. The wrapper code implementation uses the Platform Invoke (PINVOKE) interface to access natively compiled C/C++ code. The PINVOKE interface has been chosen over Microsoft's Managed C++ interface as it is portable to both Microsoft Windows and non-Microsoft platforms. PINVOKE is part of the ECMA/ISO C# specification. Swig C# works equally well on non-Microsoft operating systems such as Linux, Solaris and Apple Mac using Mono and Portable.NET.

The C# module is very similar to the Java module, so until some documentation has been written, please use the [Java documentation](#) as a guide to using SWIG with C#. The rest of this chapter should be read in conjunction with the Java documentation as it lists the main differences.

Director support (virtual method callbacks into C#) has not yet been implemented and is the main missing feature compared to Java. Less of the STL is supported and there are also a few minor utility typemaps in the various.i library which are missing.

The most notable differences to Java are the following:

- When invoking SWIG use the `-csharp` command line option instead of `-java`.
- The `-package` command line option does not exist.
- The `-namespace <name>` commandline option will generate all code into the namespace specified by `<name>`.
- The `-dllimport <name>` commandline option specifies the name of the DLL for the `DllImport` attribute for every `PInvoke` method. If this commandline option is not given, the `DllImport` DLL name is the same as the module name. This option is useful for when one wants to invoke SWIG multiple times on different modules, yet compile all the resulting code into a single DLL.
- C/C++ variables are wrapped with C# properties and not JavaBean style getters and setters.
- Global constants are generated into the module class. There is no constants interface.
- There is no implementation for type unsafe enums – not deemed necessary.
- The default enum wrapping approach is proper C# enums, not typesafe enums.  
Note that `%cconst(0)` will be ignored when wrapping C/C++ enums with proper C# enums. This is because C# enum items must be initialised from a compile time constant. If an enum item has an initialiser and the initialiser doesn't compile as C# code, then the `%csconstvalue` directive must be used as `%cconst(0)` will have no effect. If it was used, it would generate an illegal runtime initialisation via a `PINVOKE` call.
- C# doesn't support the notion of throws clauses. Therefore there is no 'throws' typemap attribute support for adding exception classes to a throws clause. Likewise there is no need for an equivalent to `%javaexception`.
- Typemap equivalent names:

<code>jni</code>	<code>-&gt; ctype</code>
<code>jtype</code>	<code>-&gt; imtype</code>
<code>jstype</code>	<code>-&gt; cstype</code>
<code>javain</code>	<code>-&gt; csin</code>
<code>javaout</code>	<code>-&gt; csout</code>
<code>javainterfaces</code>	<code>-&gt; csinterfaces and csinterfaces_derived</code>
<code>javabase</code>	<code>-&gt; csbase</code>
<code>javaclassmodifiers</code>	<code>-&gt; csclassmodifiers</code>
<code>javacode</code>	<code>-&gt; cscode</code>
<code>javainports</code>	<code>-&gt; csimports</code>
<code>javabody</code>	<code>-&gt; csbody</code>
<code>javafinalize</code>	<code>-&gt; csfinalize</code>
<code>javadestruct</code>	<code>-&gt; csdestruct</code>
<code>javadestruct_derived</code>	<code>-&gt; csdestruct_derived</code>

- Additional typemaps:

<code>csvarin</code>	C# code property set typemap
<code>csvarout</code>	C# code property get typemap

- Feature equivalent names:

<code>%javaconst</code>	<code>-&gt; %csconst</code>
<code>%javaconstvalue</code>	<code>-&gt; %csconstvalue</code>
<code>%javamethodmodifiers</code>	<code>-&gt; %csmethodmodifiers</code>

- Pragma equivalent names:

<code>%pragma ( java )</code>	<code>-&gt; %pragma ( csharp )</code>
<code>jniclassbase</code>	<code>-&gt; imclassbase</code>
<code>jniclassclassmodifiers</code>	<code>-&gt; imclassclassmodifiers</code>
<code>jniclasscode</code>	<code>-&gt; imclasscode</code>
<code>jniclassimports</code>	<code>-&gt; imclassimports</code>
<code>jniclassinterfaces</code>	<code>-&gt; imclassinterfaces</code>

- Special variable equivalent names:

<code>\$javaclassname</code>	<code>-&gt; \$csclassname</code>
<code>\$javainput</code>	<code>-&gt; \$csinput</code>
<code>\$jnical</code>	<code>-&gt; \$imcall</code>

### **\$dllimport**

This is a C# only special variable that can be used in typemaps, pragmas, features etc. The special variable will get translated into the value specified by the `-dllimport` commandline option if specified, otherwise it is equivalent to the **\$module** special variable.

The intermediary classname has PINVOKE appended after the module name instead of JNI, for example `modulenamePINVOKE`.

The directory `Examples/csharp` has a number of simple examples. Visual Studio .NET 2003 solution and project files are available for compiling with the Microsoft .NET C# compiler on Windows. If your SWIG installation went well on a Unix environment and your C# compiler was detected, you should be able to type `make` in each example directory, then `ilrun runme` (Portable.NET C# compiler) or `mono runme` (Mono C# compiler) to run the examples. Windows users can also get the examples working using a [Cygwin](#) or [MinGW](#) environment for automatic configuration of the example makefiles. Any one of the three C# compilers (Portable.NET, Mono or Microsoft) can be detected from within a Cygwin or Mingw environment if installed in your path.

# 17 SWIG and Chicken

- [Preliminaries](#)
  - ◆ [Running SWIG in C mode](#)
  - ◆ [Running SWIG in C++ mode](#)
- [Code Generation](#)
  - ◆ [Naming Conventions](#)
  - ◆ [Modules](#)
  - ◆ [Constants and Variables](#)
  - ◆ [Functions](#)
- [TinyCLOS](#)
- [Compilation](#)
- [Linkage](#)
  - ◆ [Shared library](#)
  - ◆ [Static binary](#)
- [Typemaps](#)
- [Pointers](#)
- [Unsupported features and known problems](#)

This chapter describes SWIG's support of CHICKEN. CHICKEN is a Scheme-to-C compiler supporting most of the language features as defined in the *Revised<sup>5</sup> Report on Scheme*. Its main attributes are that it

1. generates portable C code
2. includes a customizable interpreter
3. links to C libraries with a simple Foreign Function Interface
4. supports full tail-recursion and first-class continuations

When confronted with a large C library, CHICKEN users can use SWIG to generate CHICKEN wrappers for the C library. However, the real advantages of using SWIG with CHICKEN are its *support for C++* — object-oriented code is difficult to wrap by hand in CHICKEN — and its *typed pointer representation*, essential for C and C++ libraries involving structures or classes.

## 17.1 Preliminaries

CHICKEN support was introduced to SWIG in version 1.3.18. SWIG relies on some recent additions to CHICKEN, which are only present in releases of CHICKEN with version number *greater than or equal to 1.40*.

CHICKEN can be downloaded from <http://www.call-with-current-continuation.org/>. You may want to look at any of the examples in Examples/chicken/ or Examples/GIFPlot/Chicken for the basic steps to run SWIG CHICKEN. We will generically refer to the *wrapper* as the generated files.

### 17.1.1 Running SWIG in C mode

To run SWIG CHICKEN in C mode, use the `-chicken` option.

```
% swig -chicken example.i
```

To allow the wrapper to take advantage of future CHICKEN code generation improvements, part of the wrapper is direct CHICKEN function calls (`example_wrap.c`) and part is CHICKEN Scheme (`example.scm`). The basic Scheme code must be compiled to C using your system's CHICKEN compiler.

```
% chicken example.scm -output-file oexample.c
```

So for the C mode of SWIG CHICKEN, `example_wrap.c` and `oexample.c` are the files that must be compiled to object files and linked into your project.



## 17.1.2 Running SWIG in C++ mode

To run SWIG CHICKEN in C++ mode, use the `-chicken -c++` option.

```
% swig -chicken -c++ example.i
```

This will generate `example_wrap.cxx` and `example.scm`. The basic Scheme code must be compiled to C using your system's CHICKEN compiler.

```
% chicken example.scm -output-file oexample.c
```

So for the C++ mode of SWIG CHICKEN, `example_wrap.cxx` and `oexample.c` are the files that must be compiled to object files and linked into your project.

## 17.2 Code Generation

### 17.2.1 Naming Conventions

Given a C variable, function or constant declaration named `Foo_Bar`, the declaration will be available in CHICKEN as an identifier ending with `Foo-Bar`. That is, an underscore is converted to a dash.

You may control what the CHICKEN identifier will be by using the `%rename` SWIG directive in the SWIG interface file.

### 17.2.2 Modules

The name of the module must be declared one of two ways:

- Placing `%module example` in the SWIG interface file.
- Using `-module example` on the SWIG command line.

The generated `example.scm` file then exports `(declare (unit modulename))`

CHICKEN will be able to access the module using the `(declare (uses modulename))` CHICKEN Scheme form.

### 17.2.3 Constants and Variables

Constants may be created using any of the four constructs in the interface file:

```
1. #define MYCONSTANT1 ...
2. %constant int MYCONSTANT2 = ...
3. const int MYCONSTANT3 = ...
4. enum { MYCONSTANT4 = ... };
```

In all cases, the constants may be accessed from with CHICKEN using the form `(MYCONSTANT1)`; that is, the constants may be accessed using the read-only parameter form.

Variables are accessed using the full parameter form. For example, to set the C variable `"int my_variable;"`, use the Scheme form `(my-variable 2345)`. To get the C variable, use `(my-variable)`.

### 17.2.4 Functions

C functions declared in the SWIG interface file will have corresponding CHICKEN Scheme procedures. For example, the C function `"int sqrt(double x);"` will be available using the Scheme form `(sqrt 2345.0)`. A void return value will give `C_SCHEME_UNDEFINED` as a result.

A function may return more than one value by using the `OUTPUT` specifier (see `Lib/chicken/typemaps.i`). They will be returned as a Scheme list if there is more than one result (that is, a non-void return value and at least one argout parameter, or a void return value and at least two argout parameters).

## 17.3 TinyCLOS

The author of TinyCLOS, Gregor Kiczales, describes TinyCLOS as:

Tiny CLOS is a Scheme implementation of a 'kernelized' CLOS, with a metaobject protocol. The implementation is even simpler than the simple CLOS found in 'The Art of the Metaobject Protocol,' weighing in at around 850 lines of code, including (some) comments and documentation.

Almost all good Scheme books describe how to use metaobjects and generic procedures to implement an object-oriented Scheme system. Please consult a Scheme book if you are unfamiliar with the concept.

CHICKEN has a modified version of TinyCLOS, which SWIG CHICKEN uses if the `-proxy` argument is given. If `-proxy` is passed, then the generated `example.scm` file will contain TinyCLOS class definitions. A class named `Foo` is declared as `<Foo>`, and each member variable is allocated a slot. Member functions are exported as generic functions.

Primitive symbols and functions (the interface that would be presented if `-proxy` was not passed) are hidden and no longer accessible. If the `-unhideprimitive` command line argument is passed to SWIG, then the primitive symbols will be available, but each will be prefixed by the string "primitive:"

The exported symbol names can be controlled with the `-closprefix` and `-useclassprefix` arguments. If `-useclassprefix` is passed to SWIG, every member function will be generated with the class name as a prefix. If the `-closprefix mymod:` argument is passed to SWIG, then the exported functions will be prefixed by the string "mymod:". If `-useclassprefix` is passed, `-closprefix` is ignored.

## 17.4 Compilation

Please refer to *CHICKEN – A practical and portable Scheme system – User's manual* for detailed help on how to compile C code for use in a CHICKEN program. Briefly, to compile C code, be sure to add ``chicken-config -cflags`` or ``chicken-config -shared -cflags`` to your compiler options. Use the `-shared` option if you want to create a dynamically loadable module. You might also want to use the much simpler `csc` or `csc.bat`.

## 17.5 Linkage

Please refer to *CHICKEN – A practical and portable Scheme system – User's manual* for detailed help on how to link object files to create a CHICKEN Scheme program. Briefly, to link object files, be sure to add ``chicken-config -extra-libs -libs`` or ``chicken-config -shared -extra-libs -libs`` to your linker options. Use the `-shared` option if you want to create a dynamically loadable module.

### 17.5.1 Shared library

The easiest way to use SWIG and CHICKEN is to use the `csc` compiler wrapper provided by CHICKEN. Assume you have a SWIG interface file in `example.i` and the C functions being wrapped are in `example_impl.c`.

```
$ swig -chicken example.i
$ csc -svk example.scm example_impl.c example_wrap.c
$ csi example.so test_script.scm
```

You must be careful not to name the `example_impl.c` file `example.c` because when compiling `example.scm`, `csc` compiles that into `example.c`!

The `test_script.scm` should have `(load-library 'example "example.so")` and `(declare (uses example))`.

As well, the path to `example.so` should be accessible to the loader. You might need to set `LD_LIBRARY_PATH`.

## 17.5.2 Static binary

Again, we can easily use `csc` to build a binary.

```
$ swig -chicken example.i
$ csc -vk example.scm example_impl.c example_wrap.c test_script.scm -o example
$ ./example
```

## 17.6 Typemaps

The Chicken module handles all types via typemaps. This information is read from `Lib/chicken/typemaps.i` and `Lib/chicken/chicken.swg`.

## 17.7 Pointers

For pointer types, SWIG uses CHICKEN tagged pointers. A tagged pointer is an ordinary CHICKEN pointer with an extra slot for a void \*. With SWIG CHICKEN, this void \* is a pointer to a type-info structure. So each pointer used as input or output from the SWIG-generated CHICKEN wrappers will have type information attached to it. This will let the wrappers correctly determine which method should be called according to the object type hierarchy exposed in the SWIG interface files.

To construct a Scheme object from a C pointer, the wrapper code calls the function `SWIG_NewPointerObj(void *ptr, swig_type_info *type, int owner)`. The function that calls `SWIG_NewPointerObj` must have a variable declared `C_word *known_space = C_alloc(C_SIZEOF_SWIG_POINTER)`; It is ok to call `SWIG_NewPointerObj` more than once, just make sure `known_space` has enough space for all the created pointers.

To get the pointer represented by a CHICKEN tagged pointer, the wrapper code calls the function `SWIG_ConvertPtr(C_word s, void **result, swig_type_info *type, int flags)`, passing a pointer to a struct representing the expected pointer type.

## 17.8 Unsupported features and known problems

- No exception handling.
- No director support.
- No support for c++ standard types like `std::vector`.
- No support for automatic garbage collection of wrapped classes and structures. (Planned on adding in SWIG version 1.3.24)
- Importing multiple SWIG modules not working with TinyCLOS. (Planned on fixing for 1.3.24)
- Problems with complicated function overloading. (Planned on fixing for 1.3.24)

# 18 SWIG and Guile

- [Meaning of "Module"](#)
- [Using the SCM or GH Guile API](#)
- [Linkage](#)
  - ◆ [Simple Linkage](#)
  - ◆ [Passive Linkage](#)
  - ◆ [Native Guile Module Linkage](#)
  - ◆ [Old Auto-Loading Guile Module Linkage](#)
  - ◆ [Hobbit4D Linkage](#)
- [Underscore Folding](#)
- [Typemaps](#)
- [Representation of pointers as smobs](#)
  - ◆ [GH Smobs](#)
  - ◆ [SCM Smobs](#)
  - ◆ [Garbage Collection](#)
- [Exception Handling](#)
- [Procedure documentation](#)
- [Procedures with setters](#)
- [GOOPS Proxy Classes](#)
  - ◆ [Naming Issues](#)
  - ◆ [Linking](#)

This section details guile-specific support in SWIG.

## 18.1 Meaning of "Module"

There are three different concepts of "module" involved, defined separately for SWIG, Guile, and Libtool. To avoid horrible confusion, we explicitly prefix the context, e.g., "guile-module".

## 18.2 Using the SCM or GH Guile API

The guile module can currently export wrapper files that use the guile GH interface or the SCM interface. This is controlled by an argument passed to swig. The "-gh" argument causes swig to output GH code, and the "-scm" argument causes swig to output SCM code. Right now the "-scm" argument is the default. The "-scm" wrapper generation assumes a guile version  $\geq 1.6$  and has several advantages over the "-gh" wrapper generation including garbage collection and GOOPS support. The "-gh" wrapper generation can be used for older versions of guile. The guile GH wrapper code generation is deprecated and the SCM interface is the default. The SCM and GH interface differ greatly in how they store pointers and have completely different run-time code. See below for more info.

The GH interface to guile is deprecated. Read more about why in the [Guile manual](#). The idea of the GH interface was to provide a high level API that other languages and projects could adopt. This was a good idea, but didn't pan out well for general development. But for the specific, minimal uses that the SWIG typemaps put the GH interface to use is ideal for using a high level API. So even though the GH interface is deprecated, SWIG will continue to use the GH interface and provide mappings from the GH interface to whatever API we need. We can maintain this mapping where guile failed because SWIG uses a small subset of all the GH functions which map easily. All the guile typemaps like `typemaps.i` and `std_vector.i` will continue to use the GH functions to do things like create lists of values, convert strings to integers, etc. Then every language module will define a mapping between the GH interface and whatever custom API the language uses. This is currently implemented by the guile module to use the SCM guile API rather than the GH guile API. For example, here are some of the current mapping file for the SCM API

```
#define gh_append2(a, b) scm_append(scm_listify(a, b, SCM_UNDEFINED))
#define gh_apply(a, b) scm_apply(a, b, SCM_EOL)
#define gh_bool2scm SCM_BOOL
#define gh_boolean_p SCM_BOOLP
#define gh_car SCM_CAR
```

```
#define gh_cdr SCM_CDR
#define gh_cons scm_cons
#define gh_double2scm scm_make_real
...
```

This file is parsed by SWIG at wrapper generation time, so every reference to a `gh_` function is replaced by a `scm_` function in the wrapper file. Thus the `gh_` function calls will never be seen in the wrapper; the wrapper will look exactly like it was generated for the specific API. Currently only the guile language module has created a mapping policy from `gh_` to `scm_`, but there is no reason other languages (like `mzscheme` or `chicken`) couldn't also use this. If that happens, there is A LOT less code duplication in the standard typemaps.

## 18.3 Linkage

Guile support is complicated by a lack of user community cohesiveness, which manifests in multiple shared-library usage conventions. A set of policies implementing a usage convention is called a **linkage**.

### 18.3.1 Simple Linkage

The default linkage is the simplest; nothing special is done. In this case the function `SWIG_init()` is exported. Simple linkage can be used in several ways:

- **Embedded Guile, no modules.** You want to embed a Guile interpreter into your program; all bindings made by SWIG shall show up in the root module. Then call `SWIG_init()` in the `inner_main()` function. See the "simple" and "matrix" examples under `Examples/guile`.
- **Dynamic module mix-in.** You want to create a Guile module using `define-module`, containing both Scheme code and bindings made by SWIG; you want to load the SWIG modules as shared libraries into Guile.

```
(define-module (my module))
(define my-so (dynamic-link "./example.so"))
(dynamic-call "SWIG_init" my-so) ; make SWIG bindings
;; Scheme definitions can go here
```

Newer Guile versions provide a shorthand for `dynamic-link` and `dynamic-call`:

```
(load-extension "./example.so" "SWIG_init")
```

You need to explicitly export those bindings made by SWIG that you want to import into other modules:

```
(export foo bar)
```

In this example, the procedures `foo` and `bar` would be exported. Alternatively, you can export all bindings with the following module-system hack:

```
(module-map (lambda (sym var)
  (module-export! (current-module) (list sym)))
  (current-module))
```

SWIG can also generate this Scheme stub (from `define-module` up to `export`) semi-automagically if you pass it the command-line argument `-scmstub`. The code will be exported in a file called `module.scm` in the directory specified by `-outdir` or the current directory if `-outdir` is not specified. Since SWIG doesn't know how to load your extension module (with `dynamic-link` or `load-extension`), you need to supply this information by including a directive like this in the interface file:

```
%scheme %{ (load-extension "./example.so" "SWIG_init") %}
```

(The `%scheme` directive allows to insert arbitrary Scheme code into the generated file `module.scm`; it is placed between the `define-module` form and the `export` form.)

If you want to include several SWIG modules, you would need to rename `SWIG_init` via a preprocessor define to avoid symbol clashes. For this case, however, passive linkage is available.

### 18.3.2 Passive Linkage

Passive linkage is just like simple linkage, but it generates an initialization function whose name is derived from the module and package name (see below).

You should use passive linkage rather than simple linkage when you are using multiple modules.

### 18.3.3 Native Guile Module Linkage

SWIG can also generate wrapper code that does all the Guile module declarations on its own if you pass it the `-Linkage module` command-line option. This requires Guile 1.5.0 or later.

The module name is set with the `-package` and `-module` command-line options. Suppose you want to define a module with name `(my lib foo)`; then you would have to pass the options `-package my/lib -module foo`. Note that the last part of the name can also be set via the SWIG directive `%module`.

You can use this linkage in several ways:

- **Embedded Guile with SWIG modules.** You want to embed a Guile interpreter into your program; the SWIG bindings shall be put into different modules. Simply call the function `scm_init_my_modules_foo_module` in the `inner_main()` function.
- **Dynamic Guile modules.** You want to load the SWIG modules as shared libraries into Guile; all bindings are automatically put in newly created Guile modules.

```
(define my-so (dynamic-link "./foo.so"))
;; create new module and put bindings there:
(dynamic-call "scm_init_my_modules_foo_module" my-so)
```

Newer Guile versions have a shorthand procedure for this:

```
(load-extension "./foo.so" "scm_init_my_modules_foo_module")
```

### 18.3.4 Old Auto-Loading Guile Module Linkage

Guile used to support an autoloading facility for object-code modules. This support has been marked deprecated in version 1.4.1 and is going to disappear sooner or later. SWIG still supports building auto-loading modules if you pass it the `-Linkage ltdlmod` command-line option.

Auto-loading worked like this: Suppose a module with name `(my lib foo)` is required and not loaded yet. Guile will then search all directories in its search path for a Scheme file `my/modules/foo.scm` or a shared library `my/modules/libfoo.so` (or `my/modules/libfoo.la`; see the GNU libtool documentation). If a shared library is found that contains the symbol `scm_init_my_modules_foo_module`, the library is loaded, and the function at that symbol is called with no arguments in order to initialize the module.

When invoked with the `-Linkage ltdlmod` command-line option, SWIG generates an exported module initialization function with an appropriate name.

### 18.3.5 Hobbit4D Linkage

The only other linkage supported at this time creates shared object libraries suitable for use by hobbit's `(hobbit4d link) guile` module. This is called the "hobbit" linkage, and requires also using the `"-package"` command line option to set the part of the module name before the last symbol. For example, both command lines:

```
swig -guile -package my/lib foo.i
swig -guile -package my/lib -module foo foo.i
```

would create module `(my lib foo)` (assuming in the first case `foo.i` declares the module to be "foo"). The installed files are `my/lib/libfoo.so.X.Y.Z` and friends. This scheme is still very experimental; the (hobbit4d link) conventions are not well understood.

## 18.4 Underscore Folding

Underscores are converted to dashes in identifiers. Guile support may grow an option to inhibit this folding in the future, but no one has complained so far.

You can use the SWIG directives `%name` and `%rename` to specify the Guile name of the wrapped functions and variables (see CHANGES).

## 18.5 Typemaps

The Guile module handles all types via typemaps. This information is read from `Lib/guile/typemaps.i`. Some non-standard typemap substitutions are supported:

- `$descriptor` expands to a type descriptor for use with the `SWIG_NewPointerObj()` and `SWIG_ConvertPtr` functions.
- For pointer types, `$*descriptor` expands to a descriptor for the direct base type (i.e., one pointer is stripped), whereas `$basedescriptor` expands to a descriptor for the base type (i.e., all pointers are stripped).

A function returning `void` (more precisely, a function whose `out` typemap returns `SCM_UNSPECIFIED`) is treated as returning no values. In `argout` typemaps, one can use the macro `GUILLE_APPEND_RESULT` in order to append a value to the list of function return values.

Multiple values can be passed up to Scheme in one of three ways:

- *Multiple values as lists.* By default, if more than one value is to be returned, a list of the values is created and returned; to switch back to this behavior, use

```
%values_as_list;
```

- *Multiple values as vectors.* By issuing

```
%values_as_vector;
```

vectors instead of lists will be used.

- *Multiple values for multiple-value continuations. **This is the most elegant way.*** By issuing

```
%multiple_values;
```

multiple values are passed to the multiple-value continuation, as created by `call-with-values` or the convenience macro `receive`. The latter is available if you issue `(use-modules (srfi srfi-8))`. Assuming that your `divide` function wants to return two values, a quotient and a remainder, you can write:

```
(receive (quotient remainder)
  (divide 35 17)
  body...)
```

In *body*, the first result of `divide` will be bound to the variable `quotient`, and the second result to `remainder`.

See also the "multivalue" example.

## 18.6 Representation of pointers as smobs

For pointer types, SWIG uses Guile smobs. SWIG smobs print like this: `#<swig struct xyzzy * 0x1234affe>`. Two of them are equal? if and only if they have the same type and value.

To construct a Scheme object from a C pointer, the wrapper code calls the function `SWIG_NewPointerObj()`, passing a pointer to a struct representing the pointer type. The type index to store in the upper half of the CAR is read from this struct. To get the pointer represented by a smob, the wrapper code calls the function `SWIG_ConvertPtr()`, passing a pointer to a struct representing the expected pointer type. See also [The run-time type checker](#). If the Scheme object passed was not a SWIG smob representing a compatible pointer, a `wrong-type-arg` exception is raised.

### 18.6.1 GH Smobs

In earlier versions of SWIG, C pointers were represented as Scheme strings containing a hexadecimal rendering of the pointer value and a mangled type name. As Guile allows registering user types, so-called "smobs" (small objects), a much cleaner representation has been implemented now. The details will be discussed in the following.

A smob is a cons cell where the lower half of the CAR contains the smob type tag, while the upper half of the CAR and the whole CDR are available. `SWIG_Guile_Init()` registers a smob type named "swig" with Guile; its type tag is stored in the variable `swig_tag`. The upper half of the CAR store an index into a table of all C pointer types seen so far, to which new types seen are appended. The CDR stores the pointer value.

### 18.6.2 SCM Smobs

The SCM interface (using the `"-scm"` argument to `swig`) uses `common.swg`. The whole type system, when it is first initialized, creates two smobs named "swig" and "collected\_swig". The swig smob is used for non-garbage collected smobs, while the collected\_swig smob is used as described below. Each smob has the same format, which is a double cell created by `SCM_NEWSMOB2()`. The first word of data is the pointer to the object and the second word of data is the `swig_type_info` \* structure describing this type. This is a lot easier than the GH interface above because we can store a pointer to the type info structure right in the type. With the GH interface, there was not enough room in the smob to store two whole words of data so we needed to store part of the "swig\_type\_info address" in the smob tag. If a generated GOOPS module has been loaded, smobs will be wrapped by the corresponding GOOPS class.

### 18.6.3 Garbage Collection

Garbage collection is a feature of the new SCM interface, and it is automatically included if you pass the `"-scm"` flag to `swig`. Thus the swig garbage collection support requires guile >1.6. Garbage collection works like this. Every `swig_type_info` structure stores in its `clientdata` field a pointer to the destructor for this type. The destructor is the generated wrapper around the delete function. So swig still exports a wrapper for the destructor, it just does not call `scm_c_define_gsubr()` for the wrapped delete function. So the only way to delete an object is from the garbage collector, since the delete function is not available to scripts. How swig determines if a type should be garbage collected is exactly like described in [Object ownership and %newobject](#) in the SWIG manual. All typemaps use an `$owner` var, and the guile module replaces `$owner` with 0 or 1 depending on `feature:new`.

## 18.7 Exception Handling

SWIG code calls `scm_error` on exception, using the following mapping:

```
MAP(SWIG_MemoryError,      "swig-memory-error");
MAP(SWIG_IOError,          "swig-io-error");
MAP(SWIG_RuntimeError,     "swig-runtime-error");
MAP(SWIG_IndexError,       "swig-index-error");
MAP(SWIG_TypeError,        "swig-type-error");
MAP(SWIG_DivisionByZero,    "swig-division-by-zero");
MAP(SWIG_OverflowError,     "swig-overflow-error");
MAP(SWIG_SyntaxError,       "swig-syntax-error");
MAP(SWIG_ValueError,       "swig-value-error");
```



```
MAP(SWIG_SystemError, "swig-system-error");
```

The default when not specified here is to use "swig-error". See Lib/exception.i for details.

## 18.8 Procedure documentation

If invoked with the command-line option `-procdoc file`, SWIG creates documentation strings for the generated wrapper functions, describing the procedure signature and return value, and writes them to *file*. You need Guile 1.4 or later to make use of the documentation files.

SWIG can generate documentation strings in three formats, which are selected via the command-line option `-procdocformat format`:

- `guile-1.4` (default): Generates a format suitable for Guile 1.4.
- `plain`: Generates a format suitable for Guile 1.4.1 and later.
- `texinfo`: Generates texinfo source, which must be run through texinfo in order to get a format suitable for Guile 1.4.1 and later.

You need to register the generated documentation file with Guile like this:

```
(use-modules (ice-9 documentation))
(set! documentation-files
  (cons "file" documentation-files))
```

Documentation strings can be configured using the Guile-specific `typemap` argument `doc`. See Lib/guile/typemaps.i for details.

## 18.9 Procedures with setters

For global variables, SWIG creates a single wrapper procedure (*variable* :optional value), which is used for both getting and setting the value. For struct members, SWIG creates two wrapper procedures (*struct-member-get* pointer) and (*struct-member-set* pointer value).

If invoked with the command-line option `-emit-setters` (*recommended*), SWIG will additionally create procedures with setters. For global variables, the procedure-with-setter *variable* is created, so you can use (*variable*) to get the value and (*set!* (*variable*) *value*) to set it. For struct members, the procedure-with-setter *struct-member* is created, so you can use (*struct-member* pointer) to get the value and (*set!* (*struct-member* pointer) *value*) to set it.

If invoked with the command-line option `-only-setters`, SWIG will *only* create procedures with setters, i.e., for struct members, the procedures (*struct-member-get* pointer) and (*struct-member-set* pointer value) are *not* generated.

## 18.10 GOOPS Proxy Classes

SWIG can also generate classes and generic functions for use with Guile's Object-Oriented Programming System (GOOPS). GOOPS is a sophisticated object system in the spirit of the Common Lisp Object System (CLOS).

GOOPS support is only available with the new SCM interface (enabled with the `-scm` command-line option of SWIG). To enable GOOPS support, pass the `-proxy` argument to swig. This will export the GOOPS wrapper definitions into the *module.scm* file in the directory specified by `-outdir` or the current directory. GOOPS support requires either passive or module linkage.

The generated file will contain definitions of GOOPS classes mimicking the C++ class hierarchy.

Enabling GOOPS support implies `-emit-setters`.

If `-emit-slot-accessors` is also passed as an argument, then the generated file will contain accessor methods for all the slots in the classes and for global variables. The input class

```
class Foo {
public:
    Foo(int i) : a(i) {}
    int a;
    int getMultBy(int i) { return a * i; }
    Foo getFooMultBy(int i) { return Foo(a * i); }
};
Foo getFooPlus(int i) { return Foo(a + i); }
```

will produce (if `-emit-slot-accessors` is not passed as a parameter)

```
(define-class <Foo> (<swig>)
  (a #:allocation #:swig-virtual
    #:slot-ref primitive:Foo-a-get
    #:slot-set! primitive:Foo-a-set)
  #:metaclass <swig-metaclass>
  #:new-function primitive:new-Foo
)
(define-method (getMultBy (swig_smob <Foo>) i)
  (primitive:Foo-getMultBy (slot-ref swig_smob 'smob) i))
(define-method (getFooMultBy (swig_smob <Foo>) i)
  (make <Foo> #:init-smob (primitive:Foo-getFooMultBy (slot-ref swig_smob 'smob) i)))

(define-method (getFooPlus i)
  (make <Foo> #:init-smob (primitive:getFooPlus i)))

(export <Foo> getMultBy getFooMultBy getFooPlus )
```

and will produce (if `-emit-slot-accessors` is passed as a parameter)

```
(define-class <Foo> (<swig>)
  (a #:allocation #:swig-virtual
    #:slot-ref primitive:Foo-a-get
    #:slot-set! primitive:Foo-a-set
    #:accessor a)
  #:metaclass <swig-metaclass>
  #:new-function primitive:new-Foo
)
(define-method (getMultBy (swig_smob <Foo>) i)
  (primitive:Foo-getMultBy (slot-ref swig_smob 'smob) i))
(define-method (getFooMultBy (swig_smob <Foo>) i)
  (make <Foo> #:init-smob (primitive:Foo-getFooMultBy (slot-ref swig_smob 'smob) i)))

(define-method (getFooPlus i)
  (make <Foo> #:init-smob (primitive:getFooPlus i)))

(export <Foo> a getMultBy getFooMultBy getFooPlus )
```

which can then be used by this code

```
;; not using getters and setters
(define foo (make <Foo> #:args '(45)))
(slot-ref foo 'a)
(slot-set! foo 'a 3)
(getMultBy foo 4)
(define foo2 (getFooMultBy foo 7))
(slot-ref foo 'a)
(slot-ref (getFooPlus foo 4) 'a)

;; using getters and setters
(define foo (make <Foo> #:args '(45)))
(a foo)
```

```
(set! (a foo) 5)
(getMultBy foo 4)
(a (getFooMultBy foo 7))
```

Notice that constructor arguments are passed as a list after the `#:args` keyword. Hopefully in the future the following will be valid `(make <Foo> #:a 5 #:b 4)`

Also note that the order the declarations occur in the `.i` file make a difference. For example,

```
%module test

%{ #include "foo.h" %}

%inline %{
    int someFunc(Foo &a) {
        ...
    }
%}

#include "foo.h"
```

This is a valid SWIG file it will work as you think it will for primitive support, but the generated GOOPS file will be broken. Since the `someFunc` definition is parsed by SWIG before all the declarations in `foo.h`, the generated GOOPS file will contain the definition of `someFunc( )` before the definition of `<Foo>`. The generated GOOPS file would look like

```
;;...

(define-method (someFunc (swig_smob <Foo>))
  (primitive:someFunc (slot-ref swig_smob 'smob)))

;;...

(define-class <Foo> (<swig>))
  ;;...
)

;;...
```

Notice that `<Foo>` is used before it is defined. The fix is to just put the `%import "foo.h"` before the `%inline` block.

## 18.10.1 Naming Issues

As you can see in the example above, there are potential naming conflicts. The default exported accessor for the `Foo::a` variable is named `a`. The name of the wrapper global function is `getFooPlus`. If the `-useclassprefix` option is passed to swig, the name of all accessors and member functions will be prepended with the class name. So the accessor will be called `Foo-a` and the member functions will be called `Foo-getMultBy`. Also, if the `-goopsprefix goops:` argument is passed to swig, every identifier will be prefixed by `goops:`

Two guile-modules are created by SWIG. The first module contains the primitive definitions of all the wrapped functions and variables, and is located either in the `_wrap.cxx` file (with `-Linkage module`) or in the `scmstub` file (if `-Linkage passive -scmstub`). The name of this guile-module is the swig-module name (given on the command line with the `-module` argument or with the `%module` directive) concatenated with the string `"-primitive"`. For example, if `%module Test` is set in the swig interface file, the name of the guile-module in the `scmstub` or `-Linkage module` will be `Test-primitive`. Also, the `scmstub` file will be named `Test-primitive.scm`. The string "primitive" can be changed by the `-primsuffix swig` argument. So the same interface, with the `-primsuffix base` will produce a module called `Test-base`. The second generated guile-module contains all the GOOPS class definitions and is located in a file named `module.scm` in the directory specified with `-outdir` or the current directory. The name of this guile-module is the name of the swig-module (given on the command line or with the `%module` directive). In the previous example, the GOOPS definitions will be in a file named `Test.scm`.

Because of the naming conflicts, you can't in general use both the `-primitive` and the GOOPS guile-modules at the same

time. To do this, you need to rename the exported symbols from one or both guile-modules. For example,

```
(use-modules ((Test-primitive) #:renamer (symbol-prefix-proc 'primitive:)))
(use-modules ((Test) #:renamer (symbol-prefix-proc 'goops:)))
```

TODO: Renaming class name prefixes?

## 18.10.2 Linking

The guile-modules generated above all need to be linked together. GOOPS support requires either passive or module linkage. The exported GOOPS guile-module will be the name of the swig-module and should be located in a file called *Module.scm*. This should be installed on the autoloader path for guile, so that `(use-modules (Package Module))` will load everything needed. Thus, the top of the GOOPS guile-module will contain code to load everything needed by the interface (the shared library, the scmstub module, etc.). The `%goops` directive inserts arbitrary code into the generated GOOPS guile-module, and should be used to load the dependent libraries.

This breaks up into three cases

- **Passive Linkage without `-scmstub`:** Note that this linkage style has the potential for naming conflicts, since the primitive exported function and variable names are not wrapped in a guile-module and might conflict with names from the GOOPS guile-module (see above). Pass the `-goopsprefix` argument to solve this problem. If the `-exportprimitive` option is passed to SWIG the `(export . . .)` code that would be exported into the scmstub file is exported at the bottom of the generated GOOPS guile-module. The `%goops` directive should contain code to load the `.so` library.

```
%goops %{ (load-extension "./foo.so" "scm_init_my_modules_foo_module") %}
```

Produces the following code at the top of the generated GOOPS guile-module (with the `-package my/modules -module foo` command line arguments)

```
(define-module (my modules foo))

;; %goops directive goes here
(load-extension "./foo.so" "scm_init_my_modules_foo_module")

(use-modules (oop goops) (Swig common))
```

- **Passive Linkage with `-scmstub`:** Here, the name of the scmstub file should be `Module-primitive.scm` (with *primitive* replaced with whatever is given with the `-primsuffix` argument). The code to load the `.so` library should be located in the `%scheme` directive, which will then be added to the scmstub file. Swig will automatically generate the line `(use-modules (Package Module-primitive))` into the GOOPS guile-module. So if *Module-primitive.scm* is on the autoloader path for guile, the `%goops` directive can be empty. Otherwise, the `%goops` directive should contain whatever code is needed to load the *Module-primitive.scm* file into guile.

```
%scheme %{ (load-extension "./foo.so" "scm_init_my_modules_foo_module") %}
// only include the following definition if (my modules foo) cannot
// be loaded automatically
%goops %{
  (primitive-load "/path/to/foo-primitive.scm")
  (primitive-load "/path/to/Swig/common.scm")
%}
```

Produces the following code at the top of the generated GOOPS guile-module

```
(define-module (my modules foo))

;; %goops directive goes here (if any)
(primitive-load "/path/to/foo-primitive.scm")
(primitive-load "/path/to/Swig/common.scm")

(use-modules (oop goops) (Swig common))
```

## SWIG-1.3 Documentation

```
(use-modules ((my modules foo-primitive) :renamer (symbol-prefix-proc
                                                    'primitive:)))
```

- **Module Linkage:** This is very similar to passive linkage with a scmstub file. Swig will also automatically generate the line `(use-modules (Package Module-primitive))` into the GOOPS guile-module. Again the `%goops` directive should contain whatever code is needed to get that module loaded into guile.

```
%goops %{ (load-extension "./foo.so" "scm_init_my_modules_foo_module") %}
```

Produces the following code at the top of the generated GOOPS guile-module

```
(define-module (my modules foo))

;; %goops directive goes here (if any)
(load-extension "./foo.so" "scm_init_my_modules_foo_module")

(use-modules (oop goops) (Swig common))
(use-modules ((my modules foo-primitive) :renamer (symbol-prefix-proc
                                                    'primitive:)))
```

**(Swig common):** The generated GOOPS guile-module also imports definitions from the (Swig common) guile-module. This module is included with SWIG and should be installed by SWIG into the autoload path for guile (based on the configure script and whatever arguments are passed). If it is not, then the `%goops` directive also needs to contain code to load the `common.scm` file into guile. Also note that if you are trying to install the generated wrappers on a computer without SWIG installed, you will need to include the `common.swg` file along with the install.

**Multiple Modules:** Type dependencies between modules is supported. For example, if `mod1` includes definitions of some classes, and `mod2` includes some classes derived from classes in `mod1`, the generated GOOPS file for `mod2` will declare the correct superclasses. The only problem is that since `mod2` uses symbols from `mod1`, the `mod2` GOOPS file must include a `(use-modules (mod2))`. Currently, SWIG does not automatically export this line; it must be included in the `%goops` directive of `mod2`. Maybe in the future SWIG can detect dependencies and export this line. (how do other language modules handle this problem?)

# 19 SWIG and Java

- [Overview](#)
- [Preliminaries](#)
  - ◆ [Running SWIG](#)
  - ◆ [Additional Commandline Options](#)
  - ◆ [Getting the right header files](#)
  - ◆ [Compiling a dynamic module](#)
  - ◆ [Using your module](#)
  - ◆ [Dynamic linking problems](#)
  - ◆ [Compilation problems and compiling with C++](#)
  - ◆ [Building on Windows](#)
    - ◇ [Running SWIG from Visual Studio](#)
    - ◇ [Using NMAKE](#)
- [A tour of basic C/C++ wrapping](#)
  - ◆ [Modules, packages and generated Java classes](#)
  - ◆ [Functions](#)
  - ◆ [Global variables](#)
  - ◆ [Constants](#)
  - ◆ [Enumerations](#)
    - ◇ [Anonymous enums](#)
    - ◇ [Typesafe enums](#)
    - ◇ [Proper Java enums](#)
    - ◇ [Type unsafe enums](#)
    - ◇ [Simple enums](#)
  - ◆ [Pointers](#)
  - ◆ [Structures](#)
  - ◆ [C++ classes](#)
  - ◆ [C++ inheritance](#)
  - ◆ [Pointers, references, arrays and pass by value](#)
    - ◇ [Null pointers](#)
  - ◆ [C++ overloaded functions](#)
  - ◆ [C++ default arguments](#)
  - ◆ [C++ namespaces](#)
  - ◆ [C++ templates](#)
  - ◆ [C++ Smart Pointers](#)
- [Further details on the generated Java classes](#)
  - ◆ [The intermediary JNI class](#)
    - ◇ [The intermediary JNI class pragmas](#)
  - ◆ [The Java module class](#)
    - ◇ [The Java module class pragmas](#)
  - ◆ [Java proxy classes](#)
    - ◇ [Memory management](#)
    - ◇ [Inheritance](#)
    - ◇ [Proxy classes and garbage collection](#)
  - ◆ [Type wrapper classes](#)
  - ◆ [Enum classes](#)
    - ◇ [Typesafe enum classes](#)
    - ◇ [Proper Java enum classes](#)
    - ◇ [Type unsafe enum classes](#)
- [Cross language polymorphism using directors \(experimental\)](#)
  - ◆ [Enabling directors](#)
  - ◆ [Director classes](#)
  - ◆ [Overhead and code bloat](#)
  - ◆ [Simple directors example](#)

- [Common customization features](#)
  - ◆ [C/C++ helper functions](#)
  - ◆ [Class extension with %extend](#)
  - ◆ [Exception handling with %exception and %javaexception](#)
  - ◆ [Method access with %javamethodmodifiers](#)
- [Tips and techniques](#)
  - ◆ [Input and output parameters using primitive pointers and references](#)
  - ◆ [Simple pointers](#)
  - ◆ [Wrapping C arrays with Java arrays](#)
  - ◆ [Unbounded C Arrays](#)
- [Java typemaps](#)
  - ◆ [Default primitive type mappings](#)
  - ◆ [Sixty four bit JVMs](#)
  - ◆ [What is a typemap?](#)
  - ◆ [Typemaps for mapping C/C++ types to Java types](#)
  - ◆ [Java typemap attributes](#)
  - ◆ [Java special variables](#)
  - ◆ [Typemaps for both C and C++ compilation](#)
  - ◆ [Java code typemaps](#)
  - ◆ [Director specific typemaps](#)
- [Typemap Examples](#)
  - ◆ [Simpler Java enums for enums without initializers](#)
  - ◆ [Handling C++ exception specifications as Java exceptions](#)
  - ◆ [NaN Exception – exception handling for a particular type](#)
  - ◆ [Converting Java String arrays to char \\*\\*](#)
  - ◆ [Expanding a Java object to multiple arguments](#)
  - ◆ [Using typemaps to return arguments](#)
  - ◆ [Adding Java downcasts to polymorphic return types](#)
  - ◆ [Adding an equals method to the Java classes](#)
  - ◆ [Void pointers and a common Java base class](#)
- [Living with Java Directors](#)
- [Odds and ends](#)
  - ◆ [JavaDoc comments](#)
  - ◆ [Functional interface without proxy classes](#)
  - ◆ [Using your own JNI functions](#)
  - ◆ [Performance concerns and hints](#)
- [Examples](#)

This chapter describes SWIG's support of Java. It covers most SWIG features, but certain low-level details are covered in less depth than in earlier chapters.

## 19.1 Overview

The 100% Pure Java effort is a commendable concept, however in the real world programmers often either need to re-use their existing code or in some situations want to take advantage of Java but are forced into using some native (C/C++) code. The Java extension to SWIG makes it very easy to plumb in existing C/C++ code for access from Java, as SWIG writes the Java Native Interface (JNI) code for you. It is different to using the 'javah' tool as SWIG will wrap existing C/C++ code, whereas javah takes 'native' Java function declarations and creates C/C++ function prototypes. SWIG wraps C/C++ code using Java proxy classes and is very useful if you want to have access to large amounts of C/C++ code from Java. If only one or two JNI functions are needed then using SWIG may be overkill. SWIG enables a Java program to easily call into C/C++ code from Java. Historically, SWIG was not able to generate any code to call into Java code from C++. However, SWIG now supports full cross language polymorphism and code is generated to call up from C++ to Java when wrapping C++ virtual methods.

Java is one of the few non-scripting language modules in SWIG. As SWIG utilizes the type safety that the Java language offers, it takes a somewhat different approach to that used for scripting languages. In particular runtime type checking and the runtime library are not used by Java. This should be borne in mind when reading the rest of the SWIG documentation. This chapter on

Java is relatively self contained and will provide you with nearly everything you need for using SWIG and Java. However, the "[SWIG Basics](#)" chapter will be a useful read in conjunction with this one.

This chapter starts with a few practicalities on running SWIG and compiling the generated code. If you are looking for the minimum amount to read, have a look at the sections up to and including the [tour of basic C/C++ wrapping](#) section which explains how to call the various C/C++ code constructs from Java. Following this section are details of the C/C++ code and Java classes that SWIG generates. Due to the complexities of C and C++ there are different ways in which C/C++ code could be wrapped and called from Java. SWIG is a powerful tool and the rest of the chapter details how the default code wrapping can be tailored. Various customisation tips and techniques using SWIG directives are covered. The latter sections cover the advanced techniques of using typemaps for complete control of the wrapping process.

## 19.2 Preliminaries

SWIG 1.1 works with JDKs from JDK 1.1 to JDK1.4 (Java 2 SDK1.4) and should also work with any later versions. Given the choice, you should probably use the latest version of Sun's JDK. The SWIG Java module is known to work using Sun's JVM on Solaris, Linux and the various flavours of Microsoft Windows including Cygwin. The Kaffe JVM is known to give a few problems and at the time of writing was not a fully fledged JVM with full JNI support. The generated code is also known to work on vxWorks using WindRiver's PJava 3.1. The best way to determine whether your combination of operating system and JDK will work is to test the examples and test-suite that comes with SWIG. Run `make -k check` from the SWIG root directory after installing SWIG on Unix systems.

The Java module requires your system to support shared libraries and dynamic loading. This is the commonly used method to load JNI code so your system will more than likely support this.

### 19.2.1 Running SWIG

Suppose that you defined a SWIG module such as the following:

```
%module example
%{
#include "header.h"
%}
int fact(int n);
```

To build a Java module, run SWIG using the `-java` option :

```
%swig -java example.i
```

If building C++, add the `-c++` option:

```
$ swig -c++ -java example.i
```

This creates two different files; a C/C++ source file `example_wrap.c` or `example_wrap.cxx` and numerous Java files. The generated C/C++ source file contains the JNI wrapper code that needs to be compiled and linked with the rest of your C/C++ application.

The name of the wrapper file is derived from the name of the input file. For example, if the input file is `example.i`, the name of the wrapper file is `example_wrap.c`. To change this, you can use the `-o` option. It is also possible to change the [output directory](#) that the Java files are generated into using `-outdir`.

### 19.2.2 Additional Commandline Options

The following table list the additional commandline options available for the Java module. They can also be seen by using:

```
swig -java -help
```



**Java specific options**

- `-package <name>`      set name of the Java package to `<name>`
- `-noproxy`              generate the low-level functional interface instead of proxy classes

Their use will become clearer by the time you have finished reading this section on SWIG and Java.

**19.2.3 Getting the right header files**

In order to compile the C/C++ wrappers, the compiler needs the `jni.h` and `jni_md.h` header files which are part of the JDK. They are usually in directories like this:

```
/usr/java/include
/usr/java/include/<operating_system>
```

The exact location may vary on your machine, but the above locations are typical.

**19.2.4 Compiling a dynamic module**

The JNI code exists in a dynamic module or shared library (DLL on Windows) and gets loaded by the JVM. To build a shared library file, you need to compile your module in a manner similar to the following (shown for Solaris):

```
$ swig -java example.i
$ gcc -c example_wrap.c -I/usr/java/include -I/usr/java/include/solaris
$ ld -G example_wrap.o -o libexample.so
```

The exact commands for doing this vary from platform to platform. However, SWIG tries to guess the right options when it is installed. Therefore, you may want to start with one of the examples in the `Examples/java` directory. If that doesn't work, you will need to read the man-pages for your compiler and linker to get the right set of options. You might also check the [SWIG Wiki](#) for additional information.

The name of the shared library output file is important. If the name of your SWIG module is "example", the name of the corresponding shared library file should be "libexample.so" (or equivalent depending on your machine, see [Dynamic linking problems](#) for more information). The name of the module is specified using the `%module` directive or `-module` command line option.

**19.2.5 Using your module**

To load your shared native library module in Java, simply use Java's `System.loadLibrary` method in a Java class:

```
// main.java

public class main {
    static {
        System.loadLibrary("example");
    }

    public static void main(String argv[]) {
        System.out.println(example.fact(4));
    }
}
```

Compile all the Java files and run:

```
$ javac *.java
$ java main
24
$
```

If it doesn't work have a look at the following section which discusses problems loading the shared library.

## 19.2.6 Dynamic linking problems

As shown in the previous section the code to load a native library (shared library) is `System.loadLibrary("name")`. This can fail with an `UnsatisfiedLinkError` exception and can be due to a number of reasons.

You may get an exception similar to this:

```
$ java main
Exception in thread "main" java.lang.UnsatisfiedLinkError: no example in java.library.path
    at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1312)
    at java.lang.Runtime.loadLibrary0(Runtime.java:749)
    at java.lang.System.loadLibrary(System.java:820)
    at main.<clinit>(main.java:5)
```

The most common cause for this is an incorrect naming of the native library for the name passed to the `loadLibrary` function. The string passed to the `loadLibrary` function must not include the file extension name in the string, that is `.dll` or `.so`. The string must be *name* and not *libname* for all platforms. On Windows the native library must then be called *name.dll* and on most Unix systems it must be called *libname.so*. If you are debugging using `java -debug`, then the native library must be called *name\_g.dll* on Windows and *libname\_g.so* on Unix.

Another common reason for the native library not loading is because it is not in your path. On Windows make sure the *path* environment variable contains the path to the native library. On Unix make sure that your *LD\_LIBRARY\_PATH* contains the path to the native library. Adding paths to *LD\_LIBRARY\_PATH* can slow down other programs on your system so you may want to consider alternative approaches. For example you could recompile your native library with extra path information using `-rpath` if you're using GNU, see the GNU linker documentation (`ld` man page). You could use a command such as `ldconfig` (Linux) or `crle` (Solaris) to add additional search paths to the default system configuration (this requires root access and you will need to read the man pages).

The native library will also not load if there are any unresolved symbols in the compiled C/C++ code. The following exception is indicative of this:

```
$ java main
Exception in thread "main" java.lang.UnsatisfiedLinkError: libexample.so: undefined
symbol: fact
    at java.lang.ClassLoader$NativeLibrary.load(Native Method)
    at java.lang.ClassLoader.loadLibrary0(ClassLoader.java, Compiled Code)
    at java.lang.ClassLoader.loadLibrary(ClassLoader.java, Compiled Code)
    at java.lang.Runtime.loadLibrary0(Runtime.java, Compiled Code)
    at java.lang.System.loadLibrary(System.java, Compiled Code)
    at main.<clinit>(main.java:5)
$
```

This error usually indicates that you forgot to include some object files or libraries in the linking of the native library file. Make sure you compile both the SWIG wrapper file and the code you are wrapping into the native library file. Also make sure you pass all of the required libraries to the linker. The `java -verbose:jni` commandline switch is also a great way to get more information on unresolved symbols. One last piece of advice is to beware of the common faux pas of having more than one native library version in your path.

In summary, ensure that you are using the correct C/C++ compiler and linker combination and options for successful native library loading. If you are using the examples that ship with SWIG, then the Examples/Makefile must have these set up correctly for your system. The SWIG installation package makes a best attempt at getting these correct but does not get it right 100% of the time. The [SWIG Wiki](#) also has some settings for commonly used compiler and operating system combinations. The following section also contains some C++ specific linking problems and solutions.

## 19.2.7 Compilation problems and compiling with C++

On most machines, shared library files should be linked using the C++ compiler. For example:

```
% swig -c++ -java example.i
% g++ -c -fpic example.cxx
% g++ -c -fpic example_wrap.cxx -I/usr/java/j2sdk1.4.1/include -I/usr/java/
j2sdk1.4.1/include/linux
% g++ -shared example.o example_wrap.o -o libexample.so
```

In addition to this, you may need to include additional library files to make it work. For example, if you are using the Sun C++ compiler on Solaris, you often need to add an extra library `-lCrun` like this:

```
% swig -c++ -java example.i
% CC -c example.cxx
% CC -c example_wrap.cxx -I/usr/java/include -I/usr/java/include/solaris
% CC -G example.o example_wrap.o -L/opt/SUNWspro/lib -o libexample.so -lCrun
```

If you aren't entirely sure about the linking for C++, you might look at an existing C++ program. On many Unix machines, the `ldd` command will list library dependencies. This should give you some clues about what you might have to include when you link your shared library. For example:

```
$ ldd swig
      libstdc++-libc6.1-1.so.2 => /usr/lib/libstdc++-libc6.1-1.so.2 (0x40019000)
      libm.so.6 => /lib/libm.so.6 (0x4005b000)
      libc.so.6 => /lib/libc.so.6 (0x40077000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

Finally make sure the version of JDK header files matches the version of Java that you are running as incompatibilities could lead to compilation problems or unpredictable behaviour.

## 19.2.8 Building on Windows

Building on Windows is roughly similar to the process used with Unix. You will want to produce a DLL that can be loaded by the Java Virtual Machine. This section covers the process of using SWIG with Microsoft Visual C++ 6 although the procedure may be similar with other compilers. In order for everything to work, you will need to have a JDK installed on your machine in order to read the JNI header files.

### 19.2.8.1 Running SWIG from Visual Studio

If you are developing your application within Microsoft Visual studio, SWIG can be invoked as a custom build option. The `Examples\java` directory has a few [Windows Examples](#) containing Visual Studio project (.dsp) files. The process to re-create the project files for a C project are roughly:

- Open up a new workspace and use the AppWizard to select a DLL project.
- Add both the SWIG interface file (the .i file), any supporting C files, and the name of the wrapper file that will be created by SWIG (ie. `example_wrap.c`). Don't worry if the wrapper file doesn't exist yet—Visual Studio will keep a reference to it.
- Select the SWIG interface file and go to the settings menu. Under settings, select the "Custom Build" option.
- Enter "SWIG" in the description field.
- Enter "`swig -java -o $(ProjDir)\$(InputName)_wrap.c $(InputPath)`" in the "Build command(s) field"
- Enter "`$(ProjDir)\$(InputName)_wrap.c`" in the "Output files(s) field".
- Next, select the settings for the entire project and go to C/C++ tab and select the Preprocessor category. Add the include directories to the JNI header files under "Additional include directories", eg "`C:\jdk1.3\include,C:\jdk1.3\include\win32`".
- Next, select the settings for the entire project and go to Link tab and select the General category. Set the name of the output file to match the name of your Java module (ie. `example.dll`).

- Next, select the `example.c` and `example_wrap.c` files and go to the C/C++ tab and select the Precompiled Headers tab in the project settings. Disabling precompiled headers for these files will overcome any precompiled header errors while building.
- Finally, add the java compilation as a post build rule in the Post-build step tab in project settings, eg, "c:\jdk1.3\bin\javac \*.java"
- Build your project.

Note: If using C++, choose a C++ suffix for the wrapper file, for example `example_wrap.cxx`. Use `_wrap.cxx` instead of `_wrap.c` in the instructions above and add `-c++` when invoking swig.

Now, assuming all went well, SWIG will be automatically invoked when you build your project. When doing a build, any changes made to the interface file will result in SWIG being automatically invoked to produce a new version of the wrapper file.

The Java classes that SWIG output should also be compiled into `.class` files. To run the native code in the DLL (`example.dll`), make sure that it is in your path then run your Java program which uses it, as described in the previous section. If the library fails to load have a look at [Dynamic linking problems](#).

### 19.2.8.2 Using NMAKE

Alternatively, a Makefile for use by NMAKE can be written. Make sure the environment variables for MSVC++ are available and the MSVC++ tools are in your path. Now, just write a short Makefile like this :

```
# Makefile for using SWIG and Java for C code

SRCS          = example.c
IFILE         = example
INTERFACE     = $(IFILE).i
WRAPFILE      = $(IFILE)_wrap.c

# Location of the Visual C++ tools (32 bit assumed)

TOOLS         = c:\msdev
TARGET        = example.dll
CC            = $(TOOLS)\bin\cl.exe
LINK          = $(TOOLS)\bin\link.exe
INCLUDE32     = -I$(TOOLS)\include
MACHINE       = IX86

# C Library needed to build a DLL

DLLIBC        = msvcrt.lib oldnames.lib

# Windows libraries that are apparently needed
WINLIB        = kernel32.lib advapi32.lib user32.lib gdi32.lib comdlg32.lib winspool.lib

# Libraries common to all DLLs
LIBS          = $(DLLIBC) $(WINLIB)

# Linker options
LOPT          = -debug:full -debugtype:cv /NODEFAULTLIB /RELEASE /NOLOGO \
               /MACHINE:$(MACHINE) -entry:_DllMainCRTStartup@12 -dll

# C compiler flags

CFLAGS        = /Z7 /Od /c /nologo
JAVA_INCLUDE  = -ID:\jdk1.3\include -ID:\jdk1.3\include\win32

java::
    swig -java -o $(WRAPFILE) $(INTERFACE)
    $(CC) $(CFLAGS) $(JAVA_INCLUDE) $(SRCS) $(WRAPFILE)
    set LIB=$(TOOLS)\lib
    $(LINK) $(LOPT) -out:example.dll $(LIBS) example.obj example_wrap.obj
    javac *.java
```

To build the DLL and compile the java code, run NMAKE (you may need to run `vcvars32` first). This is a pretty simplistic Makefile, but hopefully its enough to get you started. Of course you may want to make changes for it to work for C++ by adding in the `-c++` command line switch for swig and replacing `.c` with `.cxx`.

## 19.3 A tour of basic C/C++ wrapping

By default, SWIG attempts to build a natural Java interface to your C/C++ code. Functions are wrapped as functions, classes are wrapped as classes, variables are wrapped with JavaBean type getters and setters and so forth. This section briefly covers the essential aspects of this wrapping.

### 19.3.1 Modules, packages and generated Java classes

The SWIG `%module` directive specifies the name of the Java module. When you specify `%module example`, the *module name* determines the name of some of the generated files in the module. The generated code consists of a *module class* file `example.java`, an *intermediary JNI class* file, `exampleJNI.java` as well as numerous other Java *proxy class* files. Each proxy class is named after the structs, unions and classes you are wrapping. You may also get a *constants interface* file if you are wrapping any unnamed enumerations or constants, for example `exampleConstants.java`. When choosing a module name, make sure you don't use the same name as one of the generated proxy class files nor a Java keyword. Sometimes a C/C++ type cannot be wrapped by a proxy class, for example a pointer to a primitive type. In these situations a *type wrapper class* is generated. Wrapping an enum generates an *enum class*, either a proper Java enum or a Java class that simulates the enums pattern. Details of all these generated classes will unfold as you read this section.

The JNI (C/C++) code is generated into a file which also contains the module name, for example `example_wrap.cxx` or `example_wrap.c`. These C or C++ files complete the contents of the module.

The generated Java classes can be placed into a Java package by using the `-package` commandline option. This is often combined with the `-outdir` to specify a package directory for generating the Java files.

```
swig -java -package com.bloggs.swig -outdir com/bloggs/swig example.i
```

SWIG won't create the directory, so make sure it exists beforehand.

### 19.3.2 Functions

There is no such thing as a global Java function so global C functions are wrapped as static methods in the module class. For example,

```
%module example
int fact(int n);
```

creates a static function that works exactly like you think it might:

```
public class example {
    public static int fact(int n) {
        // makes call using JNI to the C function
    }
}
```

The Java class `example` is the *module class*. The function can be used as follows from Java:

```
System.out.println(example.fact(4));
```

### 19.3.3 Global variables

C/C++ global variables are fully supported by SWIG. Java does not allow the overriding of the dot operator so all variables are accessed through getters and setters. Again because there is no such thing as a Java global variable, access to C/C++ global

variables is done through static getter and setter functions in the module class.

```
// SWIG interface file with global variables
%module example
...
extern int My_variable;
extern double density;
...
```

Now in Java :

```
// Print out value of a C global variable
System.out.println("My_variable = " + example.getMy_variable());
// Set the value of a C global variable
example.setDensity(0.8442);
```

The value returned by the getter will always be up to date even if the value is changed in C. Note that the getters and setters produced follow the JavaBean property design pattern. That is the first letter of the variable name is capitalized and preceded with set or get. If you have the misfortune of wrapping two variables that differ only in the capitalization of their first letters, use %rename to change one of the variable names. For example:

```
%rename Clash RenamedClash;
float Clash;
int clash;
```

If a variable is declared as `const`, it is wrapped as a read-only variable. That is only a getter is produced.

To make ordinary variables read-only, you can use the %immutable directive. For example:

```
%immutable;
extern char *path;
%mutable;
```

The %immutable directive stays in effect until it is explicitly disabled using %mutable.

If you just want to make a specific variable immutable, supply a declaration name. For example:

```
%immutable path;
...
extern char *path;      // Read-only (due to %immutable)
```

### 19.3.4 Constants

C/C++ constants are wrapped as Java static final variables. To create a constant, use #define or the %constant directive. For example:

```
#define PI 3.14159
#define VERSION "1.0"
%constant int FOO = 42;
%constant const char *path = "/usr/local";
```

By default the generated static final variables are initialized by making a JNI call to get their value. The constants are generated into the constants interface and look like this:

```
public interface exampleConstants {
    public final static double PI = exampleJNI.get_PI();
    public final static String VERSION = exampleJNI.get_VERSION();
    public final static int FOO = exampleJNI.get_FOO();
    public final static String path = exampleJNI.get_path();
}
```

Note that SWIG has inferred the C type and used an appropriate Java type that will fit the range of all possible values for the C type. By default SWIG generates **runtime constants**. They are not **compiler constants** that can, for example, be used in a switch statement. This can be changed by using the `%javaconst(flag)` directive. It works like all the other [%feature directives](#). The default is `%javaconst(0)`. It is possible to initialize all wrapped constants from pure Java code by placing a `%javaconst(1)` **before** SWIG parses the constants. Putting it at the top of your interface file would ensure this. Here is an example:

```
%javaconst(1);
%javaconst(0) BIG;
%javaconst(0) LARGE;

#define EXPRESSION (0x100+5)
#define BIG 1000LL
#define LARGE 2000ULL
```

generates:

```
public interface exampleConstants {
    public final static int EXPRESSION = (0x100+5);
    public final static long BIG = exampleJNI.get_BIG();
    public final static java.math.BigInteger LARGE = exampleJNI.get_LARGE();
}
```

Note that SWIG has inferred the C `long long` type from `BIG` and used an appropriate Java type (`long`) as a Java `long` is the smallest sized Java type that will take all possible values for a C `long long`. Similarly for `LARGE`.

Be careful using the `%javaconst(1)` directive as not all C code will compile as Java code. For example neither the `1000LL` value for `BIG` nor `2000ULL` for `LARGE` above would generate valid Java code. The example demonstrates how you can target particular constants (`BIG` and `LARGE`) with `%javaconst`. SWIG doesn't use `%javaconst(1)` as the default as it tries to generate code that will always compile. However, using a `%javaconst(1)` at the top of your interface file is strongly recommended as the preferred compile time constants will be generated and most C constants will compile as Java code and in anycase the odd constant that doesn't can be fixed using `%javaconst(0)`.

There is an alternative directive which can be used for these rare constant values that won't compile as Java code. This is the `%javaconstvalue(value)` directive, where `value` is a Java code replacement for the C constant and can be either a string or a number. This is useful if you do not want to use either the parsed C value nor a JNI call, such as when the C parsed value will not compile as Java code and a compile time constant is required. The same example demonstrates this:

```
%javaconst(1);
%javaconstvalue("new java.math.BigInteger(\"2000\")") LARGE;
%javaconstvalue(1000) BIG;

#define EXPRESSION (0x100+5)
#define BIG 1000LL
#define LARGE 2000ULL
```

Note the string quotes for "2000" are escaped. The following is then generated:

```
public interface exampleConstants {
    public final static int EXPRESSION = (0x100+5);
    public final static long BIG = 1000;
    public final static java.math.BigInteger LARGE = new java.math.BigInteger("2000");
}
```

Note: declarations declared as `const` are wrapped as read-only variables and will be accessed using a getter as described in the previous section. They are not wrapped as constants.

**Compatibility Note:** In SWIG-1.3.19 and earlier releases, the constants were generated into the module class and the constants interface didn't exist. Backwards compatibility is maintained as the module class implements the constants interface (even though some consider this type of interface implementation to be bad practice):

```
public class example implements exampleConstants {
}
```

You thus have the choice of accessing these constants from either the module class or the constants interface, for example, `example.EXPRESSION` or `exampleConstants.EXPRESSION`. Or if you decide this practice isn't so bad and your own class implements `exampleConstants`, you can of course just use `EXPRESSION`.

## 19.3.5 Enumerations

SWIG handles both named and unnamed (anonymous) enumerations. There is a choice of approaches to wrapping named C/C++ enums. This is due to historical reasons as SWIG's initial support for enums was limited and Java did not originally have support for enums. Each approach has advantages and disadvantages and it is important for the user to decide which is the most appropriate solution. There are four approaches of which the first is the default approach based on the so called Java typesafe enum pattern. The second generates proper Java enums. The final two approaches use simple integers for each enum item. Before looking at the various approaches for wrapping named C/C++ enums, anonymous enums are considered.

### 19.3.5.1 Anonymous enums

There is no name for anonymous enums and so they are handled like constants. For example:

```
enum { ALE, LAGER=10, STOUT, PILSNER };
```

is wrapped into the constants interface, in a similar manner as constants (see previous section):

```
public interface exampleConstants {
    public final static int ALE = exampleJNI.get_ALE();
    public final static int LAGER = exampleJNI.get_LAGER();
    public final static int STOUT = exampleJNI.get_STOUT();
    public final static int PILSNER = exampleJNI.get_PILSNER();
}
```

The `%javaconst(flag)` and `%javaconst(value)` directive introduced in the previous section on constants can also be used with enums. As is the case for constants, the default is `%javaconst(0)` as not all C values will compile as Java code. However, it is strongly recommended to add in a `%javaconst(1)` directive at the top of your interface file as it is only on very rare occasions that this will produce code that won't compile under Java. Using `%javaconst(1)` will ensure compile time constants are generated, thereby allowing the enum values to be used in Java switch statements. Example usage:

```
%javaconst(1);
%javaconst(0) PILSNER;
enum { ALE, LAGER=10, STOUT, PILSNER };
```

generates:

```
public interface exampleConstants {
    public final static int ALE = 0;
    public final static int LAGER = 10;
    public final static int STOUT = LAGER+1;
    public final static int PILSNER = exampleJNI.get_PILSNER();
}
```

As in the case of constants, you can access them through either the module class or the constants interface, for example, `example.ALE` or `exampleConstants.ALE`.

### 19.3.5.2 Typesafe enums

This is the default approach to wrapping named enums. The typesafe enum pattern is a relatively well known construct to work around the lack of enums in versions of Java prior to Java 2 SDK 1.5. It basically defines a class for the enumeration and permits a limited number of final static instances of the class. Each instance equates to an enum item within the enumeration. The implementation is in the "enumtypesafe.swg" file. Let's look at an example:



## SWIG-1.3 Documentation

```
%include "enumtypesafe.swg" // optional as typesafe enums are the default
enum Beverage { ALE, LAGER=10, STOUT, PILSNER };

public final class Beverage {
    public final static Beverage ALE = new Beverage("ALE");
    public final static Beverage LAGER = new Beverage("LAGER", exampleJNI.get_LAGER());
    public final static Beverage STOUT = new Beverage("STOUT");
    public final static Beverage PILSNER = new Beverage("PILSNER");
    [... additional support methods omitted for brevity ...]
}
```

See [Typesafe enum classes](#) to see the omitted support methods. Note that the enum item with an initializer (LAGER) is initialized with the enum value obtained via a JNI call. However, as with anonymous enums and constants, use of the `%javaconst` directive is strongly recommended to change this behaviour:

```
%include "enumtypesafe.swg" // optional as typesafe enums are the default
%javaconst(1);
enum Beverage { ALE, LAGER=10, STOUT, PILSNER };
```

will generate:

```
public final class Beverage {
    public final static Beverage ALE = new Beverage("ALE");
    public final static Beverage LAGER = new Beverage("LAGER", 10);
    public final static Beverage STOUT = new Beverage("STOUT");
    public final static Beverage PILSNER = new Beverage("PILSNER");
    [... additional support methods omitted for brevity ...]
}
```

The generated code is easier to read and more efficient as a true constant is used instead of a JNI call. As is the case for constants, the default is `%javaconst(0)` as not all C values will compile as Java code. However, it is recommended to add in a `%javaconst(1)` directive at the top of your interface file as it is only on very rare occasions that this will produce code that won't compile under Java. The `%javaconstvalue(value)` directive can also be used for typesafe enums. Note that global enums are generated into a Java class within whatever package you are using. C++ enums defined within a C++ class are generated into a static final inner Java class within the Java proxy class.

Typesafe enums have their advantages over using plain integers in that they can be used in a typesafe manner. However, there are limitations. For example, they cannot be used in switch statements and serialization is an issue. Please look at the following references for further information: [Replace Enums with Classes](#) in *Effective Java Programming* on the Sun website, [Create enumerated constants in Java](#) JavaWorld article, [Java Tip 133: More on typesafe enums](#) and [Java Tip 122: Beware of Java typesafe enumerations](#) JavaWorld tips.

Note that the syntax required for using typesafe enums is the same as that for proper Java enums. This is useful during the period that a project has to support legacy versions of Java. When upgrading to J2SDK 1.5 or later, proper Java enums could be used instead, without users having to change their code. The following section details proper Java enum generation.

### 19.3.5.3 Proper Java enums

Proper Java enums were only introduced in Java 2 SDK 1.5 so this approach is only compatible with more recent versions of Java. Java enums have been designed to overcome all the limitations of both typesafe and type unsafe enums and should be the choice solution, provided older versions of Java do not have to be supported. In this approach, each named C/C++ enum is wrapped by a Java enum. Java enums, by default, do not support enums with initializers. Java enums are in many respects similar to Java classes in that they can be customised with additional methods. SWIG takes advantage of this feature to facilitate wrapping C/C++ enums that have initializers. In order to wrap all possible C/C++ enums using proper Java enums, the "enums.swg" file must be used. Let's take a look at an example.

```
%include "enums.swg"
%javaconst(1);
enum Beverage { ALE, LAGER=10, STOUT, PILSNER };
```

will generate:

```
public enum Beverage {
    ALE,
    LAGER(10),
    STOUT,
    PILSNER;
    [... additional support methods omitted for brevity ...]
}
```

See [Proper Java enum classes](#) to see the omitted support methods. The generated Java enum has numerous additional methods to support enums with initializers, such as LAGER above. Note that as with the typesafe enum pattern, enum items with initializers are by default initialized with the enum value obtained via a JNI call. However, this is not the case above as we have used the recommended `%javaconst(1)` to avoid the JNI call. The `%javaconstvalue(value)` directive covered in the [Constants](#) section can also be used for proper Java enums. The additional support methods need not be generated if none of the enum items have initializers and this is covered later in the [Simpler Java enums for enums without initializers](#) section.

#### 19.3.5.4 Type unsafe enums

In this approach each enum item in a named enumeration is wrapped as a static final integer in a class named after the C/C++ enum name. This is a commonly used pattern in Java to simulate C/C++ enums, but it is not typesafe. However, the main advantage over the typesafe enum pattern is enum items can be used in switch statements. In order to use this approach, the "enumtypeunsafe.swg" file must be used. Let's take a look at an example.

```
%include "enumtypeunsafe.swg"
%javaconst(1);
enum Beverage { ALE, LAGER=10, STOUT, PILSNER };
```

will generate:

```
public final class Beverage {
    public final static int ALE = 0;
    public final static int LAGER = 10;
    public final static int STOUT = LAGER + 1;
    public final static int PILSNER = STOUT + 1;
}
```

As is the case previously, the default is `%javaconst(0)` as not all C/C++ values will compile as Java code. However, again it is recommended to add in a `%javaconst(1)` directive. and the `%javaconstvalue(value)` directive covered in the [Constants](#) section can also be used for type unsafe enums. Note that global enums are generated into a Java class within whatever package you are using. C++ enums defined within a C++ class are generated into a static final inner Java class within the Java proxy class.

Note that unlike typesafe enums, this approach requires users to mostly use different syntax compared with proper Java enums. Thus the upgrade path to proper enums provided in J2SDK 1.5 is more painful.

#### 19.3.5.5 Simple enums

This approach is similar to the type unsafe approach. Each enum item is also wrapped as a static final integer. However, these integers are not generated into a class named after the C/C++ enum. Instead, global enums are generated into the constants interface. Also, enums defined in a C++ class have their enum items generated directly into the Java proxy class rather than an inner class within the Java proxy class. In fact, this approach is effectively wrapping the enums as if they were anonymous enums and the resulting code is as per [anonymous enums](#). The implementation is in the "enumsimple.swg" file.

**Compatibility Note:** SWIG-1.3.21 and earlier versions wrapped all enums using this approach. The type unsafe approach is preferable to this one and this simple approach is only included for backwards compatibility with these earlier versions of SWIG.

## 19.3.6 Pointers

C/C++ pointers are fully supported by SWIG. Furthermore, SWIG has no problem working with incomplete type information. Here is a rather simple interface:

```
%module example

FILE *fopen(const char *filename, const char *mode);
int fputs(const char *, FILE *);
int fclose(FILE *);
```

When wrapped, you will be able to use the functions in a natural way from Java. For example:

```
SWIGTYPE_p_FILE f = example.fopen("junk","w");
example.fputs("Hello World\n", f);
example.fclose(f);
```

C pointers in the Java module are stored in a Java long and cross the JNI boundary held within this 64 bit number. Many other SWIG language modules use an encoding of the pointer in a string. These scripting languages use the SWIG runtime type checker for dynamic type checking as they do not support static type checking by a compiler. In order to implement static type checking of pointers within Java, they are wrapped by a simple Java class. In the example above the `FILE *` pointer is wrapped with a *type wrapper class* called `SWIGTYPE_p_FILE`.

Once obtained, a type wrapper object can be freely passed around to different C functions that expect to receive an object of that type. The only thing you can't do is dereference the pointer from Java. Of course, that isn't much of a concern in this example.

As much as you might be inclined to modify a pointer value directly from Java, don't. The value is not necessarily the same as the logical memory address of the underlying object. The value will vary depending on the native byte-ordering of the platform (i.e., big-endian vs. little-endian). Most JVMs are 32 bit applications so any JNI code must also be compiled as 32 bit. The net result is pointers in JNI code are also 32 bits and are stored in the high order 4 bytes on big-endian machines and in the low order 4 bytes on little-endian machines. By design it is also not possible to manually cast a pointer to a new type by using Java casts as it is particularly dangerous especially when casting C++ objects. If you need to cast a pointer or change its value, consider writing some helper functions instead. For example:

```
%inline %{
/* C-style cast */
Bar *FooToBar(Foo *f) {
    return (Bar *) f;
}

/* C++-style cast */
Foo *BarToFoo(Bar *b) {
    return dynamic_cast<Foo*>(b);
}

Foo *IncrFoo(Foo *f, int i) {
    return f+i;
}
%}
```

Also, if working with C++, you should always try to use the new C++ style casts. For example, in the above code, the C-style cast may return a bogus result whereas the C++-style cast will return a NULL pointer if the conversion can't be performed.

## 19.3.7 Structures

If you wrap a C structure, it is wrapped by a Java class with getters and setters for access to the member variables. For example,

```
struct Vector {
    double x,y,z;
};
```

is used as follows:

```
Vector v = new Vector();
v.setX(3.5);
v.setY(7.2);
double x = v.getX();
double y = v.getY();
```

The variable setters and getters are also based on the JavaBean design pattern already covered under the Global variables section. Similar access is provided for unions and the public data members of C++ classes.

This object is actually an instance of a Java class that has been wrapped around a pointer to the C structure. This instance doesn't actually do anything—it just serves as a proxy. The pointer to the C object is held in the Java proxy class in much the same way as pointers are held by type wrapper classes. Further details about Java proxy classes are covered a little later.

`const` members of a structure are read-only. Data members can also be forced to be read-only using the `%immutable` directive. For example:

```
struct Foo {
    ...
    %immutable;
    int x;          /* Read-only members */
    char *name;
    %mutable;
    ...
};
```

When `char *` members of a structure are wrapped, the contents are assumed to be dynamically allocated using `malloc` or `new` (depending on whether or not SWIG is run with the `-c++` option). When the structure member is set, the old contents will be released and a new value created. If this is not the behavior you want, you will have to use a `typemap` (described later).

If a structure contains arrays, access to those arrays is managed through pointers. For example, consider this:

```
struct Bar {
    int x[16];
};
```

If accessed in Java, you will see behavior like this:

```
Bar b = new Bar();
SWIGTYPE_p_int x = b.getX();
```

This pointer can be passed around to functions that expect to receive an `int *` (just like C). You can also set the value of an array member using another pointer. For example:

```
Bar b = new Bar();
SWIGTYPE_p_int x = b.getX();
Bar c = new Bar();
c.setX(x);                      // Copy contents of b.x to c.x
```

For array assignment (setters not getters), SWIG copies the entire contents of the array starting with the data pointed to by `b.x`. In this example, 16 integers would be copied. Like C, SWIG makes no assumptions about bounds checking—if you pass a bad pointer, you may get a segmentation fault or access violation. Array access can be changed from this default to use Java arrays and this is covered later.

When a member of a structure is itself a structure, it is handled as a pointer. For example, suppose you have two structures like this:

```

struct Foo {
    int a;
};

struct Bar {
    Foo f;
};

```

Now, suppose that you access the `f` member of `Bar` like this:

```

Bar b = new Bar();
Foo x = b.getF();

```

In this case, `x` is a pointer that points to the `Foo` that is inside `b`. This is the same value as generated by this C code:

```

Bar b;
Foo *x = &b->f;          /* Points inside b */

```

Because the pointer points inside the structure, you can modify the contents and everything works just like you would expect. For example:

```

Bar b = new Bar();
b.getF().setA(3);    // Modify b.f.a
Foo x = b.getF();
x.setA(3);           // Modify x.a - this is the same as b.f.a

```

### 19.3.8 C++ classes

C++ classes are wrapped by Java classes as well. For example, if you have this class,

```

class List {
public:
    List();
    ~List();
    int  search(char *item);
    void insert(char *item);
    void remove(char *item);
    char *get(int n);
    int  length;
};

```

you can use it in Java like this:

```

List l = new List();
l.insert("Ale");
l.insert("Stout");
l.insert("Lager");
String item = l.get(2);
int length = l.getLength();

```

Class data members are accessed in the same manner as C structures.

Static class members are unsurprisingly wrapped as static members of the Java class:

```

class Spam {
public:
    static void foo();
    static int bar;
};

```

The static members work like any other Java static member:

```
Spam.foo();
int bar = Spam.getBar();
```

### 19.3.9 C++ inheritance

SWIG is fully aware of issues related to C++ inheritance. Therefore, if you have classes like this

```
class Foo {
...
};

class Bar : public Foo {
...
};
```

those classes are wrapped into a hierarchy of Java classes that reflect the same inheritance structure:

```
Bar b = new Bar();
Class c = b.getClass();
System.out.println(c.getSuperclass().getName());
```

will of course display:

```
Foo
```

Furthermore, if you have functions like this

```
void spam(Foo *f);
```

then the Java function `spam()` accepts instances of `Foo` or instances of any other proxy classes derived from `Foo`.

Note that Java does not support multiple inheritance so any multiple inheritance in the C++ code is not going to work. A warning is given when multiple inheritance is detected and only the first base class is used.

### 19.3.10 Pointers, references, arrays and pass by value

In C++, there are many different ways a function might receive and manipulate objects. For example:

```
void spam1(Foo *x);      // Pass by pointer
void spam2(Foo &x);      // Pass by reference
void spam3(Foo x);       // Pass by value
void spam4(Foo x[]);     // Array of objects
```

In Java, there is no detailed distinction like this—specifically, there are only instances of classes. There are no pointers nor references. Because of this, SWIG unifies all of these types together in the wrapper code. For instance, if you actually had the above functions, it is perfectly legal to do this from Java:

```
Foo f = new Foo(); // Create a Foo
example.spam1(f);  // Ok. Pointer
example.spam2(f);  // Ok. Reference
example.spam3(f);  // Ok. Value.
example.spam4(f);  // Ok. Array (1 element)
```

Similar behavior occurs for return values. For example, if you had functions like this,

```
Foo *spam5();
Foo &spam6();
Foo spam7();
```

then all three functions will return a pointer to some `Foo` object. Since the third function (`spam7`) returns a value, newly allocated memory is used to hold the result and a pointer is returned (Java will release this memory when the returned object's finalizer is run by the garbage collector).

### 19.3.10.1 Null pointers

Working with null pointers is easy. A Java `null` can be used whenever a method expects a proxy class or typewrapper class. However, it is not possible to pass null to C/C++ functions that take parameters by value or by reference. If you try you will get a `NullPointerException`.

```
example.spam1(null);    // Pointer - ok
example.spam2(null);    // Reference - NullPointerException
example.spam3(null);    // Value - NullPointerException
example.spam4(null);    // Array - ok
```

For `spam1` and `spam4` above the Java `null` gets translated into a `NULL` pointer for passing to the C/C++ function. The converse also occurs, that is, `NULL` pointers are translated into `null` Java objects when returned from a C/C++ function.

### 19.3.11 C++ overloaded functions

C++ overloaded functions, methods, and constructors are mostly supported by SWIG. For example, if you have two functions like this:

```
%module example

void foo(int);
void foo(char *c);
```

You can use them in Java in a straightforward manner:

```
example.foo(3);          // foo(int)
example.foo("Hello");    // foo(char *c)
```

Similarly, if you have a class like this,

```
class Foo {
public:
    Foo();
    Foo(const Foo &);
    ...
};
```

you can write Java code like this:

```
Foo f = new Foo();      // Create a Foo
Foo g = new Foo(f);     // Copy f
```

Overloading support is not quite as flexible as in C++. Sometimes there are methods that SWIG cannot disambiguate as there can be more than one C++ type mapping onto a single Java type. For example:

```
void spam(int);
void spam(unsigned short);
```

Here both `int` and `unsigned short` map onto a Java `int`. Here is another example:

```
void foo(Bar *b);
void foo(Bar &b);
```

If declarations such as these appear, you will get a warning message like this:

```
example.i:12: Warning(515): Overloaded method spam(unsigned short) ignored.
Method spam(int) at example.i:11 used.
```

To fix this, you either need to ignore or rename one of the methods. For example:

```
%rename(spam_short) spam(short);
...
void spam(int);
void spam(short);    // Accessed as spam_short
```

or

```
%ignore spam(short);
...
void spam(int);
void spam(short);    // Ignored
```

### 19.3.12 C++ default arguments

Any function with a default argument is wrapped by generating an additional function for each argument that is defaulted. For example, if we have the following C++:

```
%module example

void defaults(double d=10.0, int i=0);
```

The following methods are generated in the Java module class:

```
public class example {
    public static void defaults(double d, int i) { ... }
    public static void defaults(double d) { ... }
    public static void defaults() { ... }
}
```

It is as if SWIG had parsed three separate overloaded methods. The same approach is taken for static methods, constructors and member methods.

**Compatibility note:** Versions of SWIG prior to SWIG-1.3.23 wrapped these with a single wrapper method and so the default values could not be taken advantage of from Java. Further details on default arguments and how to restore this approach are given in the more general [Default arguments](#) section.

### 19.3.13 C++ namespaces

SWIG is aware of C++ namespaces, but namespace names do not appear in the module nor do namespaces result in a module that is broken up into submodules or packages. For example, if you have a file like this,

```
%module example

namespace foo {
    int fact(int n);
    struct Vector {
        double x,y,z;
    };
};
```

it works in Java as follows:

```
int f = example.fact(3);
Vector v = new Vector();
v.setX(3.4);
double y = v.getY();
```



If your program has more than one namespace, name conflicts (if any) can be resolved using `%rename`. For example:

```
%rename(Bar_spam) Bar::spam;

namespace Foo {
    int spam();
}

namespace Bar {
    int spam();
}
```

If you have more than one namespace and you want to keep their symbols separate, consider wrapping them as separate SWIG modules. Each SWIG module can be placed into a separate package.

### 19.3.14 C++ templates

C++ templates don't present a huge problem for SWIG. However, in order to create wrappers, you have to tell SWIG to create wrappers for a particular template instantiation. To do this, you use the `%template` directive. For example:

```
%module example
%{
#include "pair.h"
%}

template<class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1&, const T2&);
    ~pair();
};

%template(pairii) pair<int,int>;
```

In Java:

```
pairii p = new pairii(3,4);
int first = p.getFirst();
int second = p.getSecond();
```

Obviously, there is more to template wrapping than shown in this example. More details can be found in the [SWIG and C++](#) chapter.

### 19.3.15 C++ Smart Pointers

In certain C++ programs, it is common to use classes that have been wrapped by so-called "smart pointers." Generally, this involves the use of a template class that implements `operator->()` like this:

```
template<class T> class SmartPtr {
    ...
    T *operator->();
    ...
}
```

Then, if you have a class like this,

```
class Foo {
public:
```

```
int x;
int bar();
};
```

A smart pointer would be used in C++ as follows:

```
SmartPtr<Foo> p = CreateFoo(); // Created somehow (not shown)
...
p->x = 3; // Foo::x
int y = p->bar(); // Foo::bar
```

To wrap this in Java, simply tell SWIG about the `SmartPtr` class and the low-level `Foo` object. Make sure you instantiate `SmartPtr` using `%template` if necessary. For example:

```
%module example
...
%template(SmartPtrFoo) SmartPtr<Foo>;
...
```

Now, in Java, everything should just "work":

```
SmartPtrFoo p = example.CreateFoo(); // Create a smart-pointer somehow
p.setX(3); // Foo::x
int y = p.bar(); // Foo::bar
```

If you ever need to access the underlying pointer returned by `operator->()` itself, simply use the `__deref__()` method. For example:

```
Foo f = p.__deref__(); // Returns underlying Foo *
```

## 19.4 Further details on the generated Java classes

In the previous section, a high-level view of Java wrapping was presented. A key component of this wrapping is that structures and classes are wrapped by Java proxy classes and type wrapper classes are used in situations where no proxies are generated. This provides a very natural, type safe Java interface to the C/C++ code and fits in with the Java programming paradigm. However, a number of low-level details were omitted. This section provides a brief overview of how the proxy classes work and then covers the type wrapper classes. Finally enum classes are covered. First, the crucial intermediary JNI class is considered.

### 19.4.1 The intermediary JNI class

In the ["SWIG basics"](#) and ["SWIG and C++"](#) chapters, details of low-level structure and class wrapping are described. To summarize those chapters, if you have a global function and class like this

```
class Foo {
public:
    int x;
    int spam(int num, Foo* foo);
};
void egg(Foo* chips);
```

then SWIG transforms the class into a set of low-level procedural wrappers. These procedural wrappers essentially perform the equivalent of this C++ code:

```
Foo *new_Foo() {
    return new Foo();
}
void delete_Foo(Foo *f) {
    delete f;
}
int Foo_x_get(Foo *f) {
```

```

    return f->x;
}
void Foo_x_set(Foo *f, int value) {
    f->x = value;
}
int Foo_spam(Foo *f, int num, Foo* foo) {
    return f->spam(num, foo);
}

```

These procedural function names don't actually exist, but their functionality appears inside the generated JNI functions. The JNI functions have to follow a particular naming convention so the function names are actually:

```

JNIEXPORT jlong JNICALL Java_exampleJNI_new_1Foo(JNIEnv *jenv, jclass jcls);
JNIEXPORT void JNICALL Java_exampleJNI_delete_1Foo(JNIEnv *jenv, jclass jcls,
                                                    jlong jarg1);
JNIEXPORT void JNICALL Java_exampleJNI_set_1Foo_1x(JNIEnv *jenv, jclass jcls,
                                                    jlong jarg1, jint jarg2);
JNIEXPORT jint JNICALL Java_exampleJNI_get_1Foo_1x(JNIEnv *jenv, jclass jcls,
                                                    jlong jarg1);
JNIEXPORT jint JNICALL Java_exampleJNI_Foo_1spam(JNIEnv *jenv, jclass jcls,
                                                  jlong jarg1, jint jarg2, jlong jarg3);
JNIEXPORT void JNICALL Java_exampleJNI_egg(JNIEnv *jenv, jclass jcls, jlong jarg1);

```

For every JNI C function there has to be a static native Java function. These appear in the intermediary JNI class:

```

class exampleJNI {
    public final static native long new_Foo();
    public final static native void delete_Foo(long jarg1);
    public final static native void set_Foo_x(long jarg1, int jarg2);
    public final static native int get_Foo_x(long jarg1);
    public final static native int Foo_spam(long jarg1, int jarg2, long jarg3);
    public final static native void egg(long jarg1);
}

```

This class contains the complete Java – C/C++ interface so all function calls go via this class. As this class acts as a go-between for all JNI calls to C/C++ code from the Java [proxy classes](#), [type wrapper classes](#) and [module class](#), it is known as the intermediary JNI class.

You may notice that SWIG uses a Java long wherever a pointer or class object needs traversing the Java–C/C++ boundary. This approach leads to minimal JNI code which makes for better performance as JNI code involves a lot of string manipulation. SWIG uses Java code wherever possible as it is compiled into byte code which requires fewer string operations.

The functions in the intermediary JNI class cannot be accessed outside of its package. Access to them is gained through the module class for globals otherwise the appropriate proxy class.

The name of the intermediary JNI class can be changed from its default, that is, the module name with JNI appended after it. The module directive attribute `jniclassname` is used to achieve this:

```
%module (jniclassname="name") modulename
```

If `name` is the same as `modulename` then the module class name gets changed from `modulename` to `modulenameModule`.

#### 19.4.1.1 The intermediary JNI class pragmas

The intermediary JNI class can be tailored through the use of pragmas, but is not commonly done. The pragmas for this class are:

Pragma	Description
<code>jniclassbase</code>	Base class for the intermediary JNI class
<code>jniclassmodifiers</code>	Class modifiers and class type for the intermediary JNI class
<code>jniclasscode</code>	Java code is copied verbatim into the intermediary JNI class

<code>jniclassimports</code>	Java code, usually one or more import statements, placed before the intermediary JNI class definition
<code>jniclassinterfaces</code>	Comma separated interface classes for the intermediary JNI class

The pragma code appears in the generated intermediary JNI class where you would expect:

```
[ jniclassimports pragma ]
[ jniclassmodifiers pragma ] jniclassname extends [ jniclassbase pragma ]
                                implements [ jniclassinterfaces pragma ] {
[ jniclasscode pragma ]
... SWIG generated native methods ...
}
```

The `jniclasscode` pragma is quite useful for adding in a static block for loading the shared library / dynamic link library and demonstrates how pragmas work:

```
%pragma(java) jniclasscode=%{
    static {
        try {
            System.loadLibrary("example");
        } catch (UnsatisfiedLinkError e) {
            System.err.println("Native code library failed to load. \n" + e);
            System.exit(1);
        }
    }
}%}
```

Pragmas will take either " " or `%{ %}` as delimiters. For example, let's change the intermediary JNI class access to public.

```
%pragma(java) jniclassclassmodifiers="public class"
```

All the methods in the intermediary JNI class will then be callable outside of the package as the method modifiers are public by default.

## 19.4.2 The Java module class

All global functions and variable getters/setters appear in the module class. For our example, there is just one function:

```
public class example {
    public static void egg(Foo chips) {
        exampleJNI.egg(Foo.getCPtr(chips));
    }
}
```

The module class is necessary as there is no such thing as a global in Java so all the C globals are put into this class. They are generated as static functions and so must be accessed as such by using the module name in the static function call:

```
example.egg(new Foo());
```

The primary reason for having the module class wrapping the calls in the intermediary JNI class is to implement static type checking. In this case only a `Foo` can be passed to the `egg` function, whereas any `long` can be passed to the `egg` function in the intermediary JNI class.

### 19.4.2.1 The Java module class pragmas

The module class can be tailored through the use of pragmas, in the same manner as the intermediary JNI class. The pragmas are similarly named and are used in the same way. The complete list follows:

Pragma	Description
<code>modulebase</code>	Base class for the module class

moduleclassmodifiers	Class modifiers and class type for the module class
modulecode	Java code is copied verbatim into the module class
moduleimports	Java code, usually one or more import statements, placed before the module class definition
moduleinterfaces	Comma separated interface classes for the module class

The pragma code appears in the generated module class like this:

```
[ moduleimports pragma ]
[ modulemodifiers pragma ] modulename extends [ modulebase pragma ]
                                implements [ moduleinterfaces pragma ] {
[ modulecode pragma ]
... SWIG generated wrapper functions ...
}
```

See [The intermediary JNI class pragmas](#) section for further details on using pragmas.

### 19.4.3 Java proxy classes

A Java proxy class is generated for each structure, union or C++ class that is wrapped. The default proxy class for our previous example looks like this:

```
public class Foo {
    private long swigCPtr;
    protected boolean swigCMemOwn;

    protected Foo(long cPtr, boolean cMemoryOwn) {
        swigCMemOwn = cMemoryOwn;
        swigCPtr = cPtr;
    }

    protected static long getCPtr(Foo obj) {
        return obj.swigCPtr;
    }

    protected void finalize() {
        delete();
    }

    public void delete() {
        if(swigCPtr != 0 && swigCMemOwn) {
            exampleJNI.delete_Foo(swigCPtr);
            swigCMemOwn = false;
        }
        swigCPtr = 0;
    }

    public void setX(int x) {
        exampleJNI.set_Foo_x(swigCPtr, x);
    }

    public int getX() {
        return exampleJNI.get_Foo_x(swigCPtr);
    }

    public int spam(int num, Foo foo) {
        return exampleJNI.Foo_spam(swigCPtr, num, Foo.getCPtr(foo));
    }

    public Foo() {
        this(exampleJNI.new_Foo(), true);
    }
}
```

This class merely holds a pointer to the underlying C++ object (`swigCPtr`). It also contains all the methods in the C++ class it is proxying plus getters and setters for public member variables. These functions call the native methods in the intermediary JNI class. The advantage of having this extra layer is the type safety that the proxy class functions offer. It adds static type checking which leads to fewer surprises at runtime. For example, you can see that if you attempt to use the `spam()` function it will only compile when the parameters passed are an `int` and a `Foo`. From a user's point of view, it makes the class work as if it were a Java class:

```
Foo f = new Foo();
f.setX(3);
int y = f.spam(5, new Foo());
```

#### 19.4.3.1 Memory management

Each proxy class has an ownership flag `swigCMemOwn`. The value of this flag determines who is responsible for deleting the underlying C++ object. If set to `true`, the proxy class's finalizer will destroy the C++ object when the proxy class is garbage collected. If set to `false`, then the destruction of the proxy class has no effect on the C++ object.

When an object is created by a constructor or returned by value, Java automatically takes ownership of the result. On the other hand, when pointers or references are returned to Java, there is often no way to know where they came from. Therefore, the ownership is set to `false`. For example:

```
class Foo {
public:
    Foo();
    Foo bar1();
    Foo &bar2();
    Foo *bar2();
};
```

In Java:

```
Foo f = new Foo();    // f.swigCMemOwn = true
Foo f1 = f.bar1();    // f1.swigCMemOwn = true
Foo f2 = f.bar2();    // f2.swigCMemOwn = false
Foo f3 = f.bar3();    // f3.swigCMemOwn = false
```

This behavior for pointers and references is especially important for classes that act as containers. For example, if a method returns a pointer to an object that is contained inside another object, you definitely don't want Java to assume ownership and destroy it!

For the most part, memory management issues remain hidden. However, there are situations where you might have to manually change the ownership of an object. For instance, consider code like this:

```
class Obj {};
class Node {
    Obj *value;
public:
    void set_value(Obj *v) { value = v; }
};
```

Now, consider the following Java code:

```
Node n = new Node();    // Create a node
{
    Obj o = new Obj();    // Create an object
    n.set_value(o);        // Set value
}                          // o goes out of scope
```

In this case, the `Node n` is holding a reference to `o` internally. However, SWIG has no way to know that this has occurred. The Java proxy class still thinks that it has ownership of `o`. As `o` has gone out of scope, it could be garbage collected in which case the

C++ destructor will be invoked and `n` will then be holding a stale pointer to `o`. If you're lucky, you will only get a segmentation fault.

To work around this, the ownership flag of `o` needs changing to `false`. The ownership flag is a private member variable of the proxy class so this is not possible without some customization of the proxy class. This is achieved using a typemap to add pure Java code to the proxy class and is detailed later in the section on typemaps.

Sometimes a function will create memory and return a pointer to a newly allocated object. SWIG has no way of knowing this so by default the proxy class does not manage the returned object. However, you can tell the proxy class to manage the memory if you specify the `%newobject` directive. Consider:

```
class Obj {...};
class Factory {
public:
    static Obj *createObj() { return new Obj(); }
};
```

If we call the factory function, then we have to manually delete the memory:

```
Obj obj = Factory.createObj();    // obj.swigCMemOwn = false
...
obj.delete();
```

Now add in the `%newobject` directive:

```
%newobject Factory::createObj();

class Obj {...};
class Factory {
public:
    static Obj *createObj() { return new Obj(); }
};
```

A call to `delete()` is no longer necessary as the garbage collector will make the C++ destructor call because `swigCMemOwn` is now true.

```
Obj obj = Factory.createObj();    // obj.swigCMemOwn = true;
...
```

### 19.4.3.2 Inheritance

Java proxy classes will mirror C++ inheritance chains. For example, given the base class `Base` and its derived class `Derived`:

```
class Base {
public:
    virtual double foo();
};

class Derived : public Base {
public:
    virtual double foo();
};
```

The base class is generated much like any other proxy class seen so far:

```
public class Base {
    private long swigCPtr;
    protected boolean swigCMemOwn;

    protected Base(long cPtr, boolean cMemoryOwn) {
        swigCMemOwn = cMemoryOwn;
        swigCPtr = cPtr;
    }
}
```

```

}

protected static long getCPtr(Base obj) {
    return obj.swigCPtr;
}

protected void finalize() {
    delete();
}

public void delete() {
    if(swigCPtr != 0 && swigCMemOwn) {
        exampleJNI.delete_Base(swigCPtr);
        swigCMemOwn = false;
    }
    swigCPtr = 0;
}

public double foo() {
    return exampleJNI.Base_foo(swigCPtr);
}

public Base() {
    this(exampleJNI.new_Base(), true);
}
}

```

The Derived class extends Base mirroring the C++ class inheritance hierarchy.

```

public class Derived extends Base {
    private long swigCPtr;

    protected Derived(long cPtr, boolean cMemoryOwn) {
        super(exampleJNI.SWIGDerivedUpcast(cPtr), cMemoryOwn);
        swigCPtr = cPtr;
    }

    protected static long getCPtr(Derived obj) {
        return obj.swigCPtr;
    }

    protected void finalize() {
        delete();
    }

    public void delete() {
        if(swigCPtr != 0 && swigCMemOwn) {
            exampleJNI.delete_Derived(swigCPtr);
            swigCMemOwn = false;
        }
        swigCPtr = 0;
        super.delete();
    }

    public double foo() {
        return exampleJNI.Derived_foo(swigCPtr);
    }

    public Derived() {
        this(exampleJNI.new_Derived(), true);
    }
}

```

Note the memory ownership is controlled by the base class. However each class in the inheritance hierarchy has its own pointer value which is obtained during construction. The `SWIGDerivedUpcast()` call converts the pointer from a `Derived *` to a `Base *`. This is a necessity as C++ compilers are free to implement pointers in the inheritance hierarchy with different values.



It is of course possible to extend Base using your own Java classes. If Derived is provided by the C++ code, you could for example add in a pure Java class Extended derived from Base. There is a caveat and that is any C++ code will not know about your pure Java class Extended so this type of derivation is restricted. However, true cross language polymorphism can be achieved using the [directors](#) feature.

### 19.4.3.3 Proxy classes and garbage collection

By default each proxy class has a `delete()` and a `finalize()` method. The `finalize()` method calls `delete()` which frees any malloc'd memory for wrapped C structs or calls the C++ class destructors. The idea is for `delete()` to be called when you have finished with the C/C++ object. Ideally you need not call `delete()`, but rather leave it to the garbage collector to call it from the finalizer. The unfortunate thing is that Sun, in their wisdom, do not guarantee that the finalizers will be called. When a program exits, the garbage collector does not always call the finalizers. Depending on what the finalizers do and which operating system you use, this may or may not be a problem.

If the `delete()` call into JNI code is just for memory handling, there is not a problem when run on most operating systems, for example Windows and Unix. Say your JNI code creates memory on the heap which your finalizers should clean up, the finalizers may or may not be called before the program exits. In Windows and Unix all memory that a process uses is returned to the system on exit, so this isn't a problem. This is not the case in some operating systems like vxWorks. If however, your finalizer calls into JNI code invoking the C++ destructor which in turn releases a TCP/IP socket for example, there is no guarantee that it will be released. Note that with long running programs the garbage collector will eventually run, thereby calling any unreferenced object's finalizers.

Some not so ideal solutions are:

1. Call the `System.runFinalizersOnExit(true)` or `Runtime.getRuntime().runFinalizersOnExit(true)` to ensure the finalizers are called before the program exits. The catch is that this is a deprecated function call as the documentation says:

*This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.*

In many cases you will be lucky and find that it works, but it is not to be advocated. Have a look at [Sun's Java web site](#) and search for `runFinalizersOnExit`.

2. From jdk1.3 onwards a new function, `addShutdownHook()`, was introduced which is guaranteed to be called when your program exits. You can encourage the garbage collector to call the finalizers, for example, add this static block to the class that has the `main()` function:

```
static {
    Runtime.getRuntime().addShutdownHook(
        new Thread() {
            public void run() { System.gc(); System.runFinalization(); }
        }
    );
}
```

Although this usually works, the documentation doesn't guarantee that `runFinalization()` will actually call the finalizers. As the the shutdown hook is guaranteed you could also make a JNI call to clean up any resources that are being tracked by the C/C++ code.

3. Call the `delete()` function manually which will immediately invoke the C++ destructor. As a suggestion it may be a good idea to set the object to null so that should the object be inadvertently used again a Java null pointer exception is thrown, the alternative would crash the JVM by using a null C pointer. For example given a SWIG generated class A:

```
A myA = new A();
// use myA ...
myA.delete();
// any use of myA here would crash the JVM
myA=null;
// any use of myA here would cause a Java null pointer exception to be thrown
```

The SWIG generated code ensures that the memory is not deleted twice, in the event the finalizers get called in addition to the manual `delete()` call.

4. Write your own object manager in Java. You could derive all SWIG classes from a single base class which could track which objects have had their finalizers run, then call the rest of them on program termination. The section on [Java typemaps](#) details how to specify a pure Java base class.

### 19.4.4 Type wrapper classes

The generated type wrapper class, for say an `int *`, looks like this:

```
public class SWIGTYPE_p_int {
    private long swigCPtr;

    protected SWIGTYPE_p_int(long cPtr, boolean bFutureUse) {
        swigCPtr = cPtr;
    }

    protected SWIGTYPE_p_int() {
        swigCPtr = 0;
    }

    protected static long getCPtr(SWIGTYPE_p_int obj) {
        return obj.swigCPtr;
    }
}
```

The methods do not have public access, so by default it is impossible to do anything with objects of this class other than pass them around. The methods in the class are part of the inner workings of SWIG. If you need to mess around with pointers you will have to use some typemaps specific to the Java module to achieve this. The section on [Java typemaps](#) details how to modify the generated code.

Note that if you use a pointer or reference to a proxy class in a function then no type wrapper class is generated because the proxy class can be used as the function parameter. If however, you need anything more complicated like a pointer to a pointer to a proxy class then a typewrapper class is generated for your use.

Note that SWIG generates a type wrapper class and not a proxy class when it has not parsed the definition of a type that gets used. For example, say SWIG has not parsed the definition of `class Snazzy` because it is in a header file that you may have forgotten to use the `%include` directive on. Should SWIG parse `Snazzy *` being used in a function parameter, it will then generate a type wrapper class around a `Snazzy` pointer. Also recall from earlier that SWIG will use a pointer when a class is passed by value or by reference:

```
void spam(Snazzy *x, Snazzy &y, Snazzy z);
```

Should SWIG not know anything about `Snazzy` then a `SWIGTYPE_p_Snazzy` must be used for all 3 parameters in the `spam` function. The Java function generated is:

```
public static void spam(SWIGTYPE_p_Snazzy x, SWIGTYPE_p_Snazzy y, SWIGTYPE_p_Snazzy z) {
    ...
}
```

Note that typedefs are tracked by SWIG and the typedef name is used to construct the type wrapper class name. For example, consider the case where `Snazzy` is a typedef to an `int` which SWIG does parse:

```
typedef int Snazzy;
void spam(Snazzy *x, Snazzy &y, Snazzy z);
```

Because the typedefs have been tracked the Java function generated is:

```
public static void spam(SWIGTYPE_p_int x, SWIGTYPE_p_int y, int z) { ... }
```

## 19.4.5 Enum classes

SWIG can generate three types of enum classes. The [Enumerations](#) section discussed these but omitted all the details. The following sub-sections detail the various types of enum classes that can be generated.

### 19.4.5.1 Typesafe enum classes

The following example demonstrates the typesafe enum classes which SWIG generates:

```
%include "enumtypesafe.swg"
%javaconst(1);
enum Beverage { ALE, LAGER=10, STOUT, PILSNER };
```

The following is what SWIG generates:

```
public final class Beverage {
    public final static Beverage ALE = new Beverage("ALE");
    public final static Beverage LAGER = new Beverage("LAGER", 10);
    public final static Beverage STOUT = new Beverage("STOUT");
    public final static Beverage PILSNER = new Beverage("PILSNER");

    public final int swigValue() {
        return swigValue;
    }

    public String toString() {
        return swigName;
    }

    public static Beverage swigToEnum(int swigValue) {
        if (swigValue < swigValues.length && swigValues[swigValue].swigValue == swigValue)
            return swigValues[swigValue];
        for (int i = 0; i < swigValues.length; i++)
            if (swigValues[i].swigValue == swigValue)
                return swigValues[i];
        throw new IllegalArgumentException("No enum " + Beverage.class + " with value " +
                                         swigValue);
    }

    private Beverage(String swigName) {
        this.swigName = swigName;
        this.swigValue = swigNext++;
    }

    private Beverage(String swigName, int swigValue) {
        this.swigName = swigName;
        this.swigValue = swigValue;
        swigNext = swigValue+1;
    }

    private static Beverage[] swigValues = { ALE, LAGER, STOUT, PILSNER };
    private static int swigNext = 0;
    private final int swigValue;
    private final String swigName;
}
```

As can be seen, there a fair number of support methods for the typesafe enum pattern. The typesafe enum pattern involves creating a fixed number of static instances of the enum class. The constructors are private to enforce this. Two constructors are available – one for C/C++ enums with an initializer and one for those without an initializer. In order to use one of these typesafe enums, the `swigToEnum` static method must be called to return a reference to one of the static instances. The JNI layer returns the enum value from the C/C++ world as an integer and this method is used to find the appropriate Java enum static instance. The `swigValue` method is used for marshalling in the other direction. The `toString` method is overridden so that the enum name is available.

### 19.4.5.2 Proper Java enum classes

The following example demonstrates the Java enums approach:

```
%include "enums.swg"
%javaconst(1);
enum Beverage { ALE, LAGER=10, STOUT, PILSNER };
```

SWIG will generate the following Java enum:

```
public enum Beverage {
    ALE,
    LAGER(10),
    STOUT,
    PILSNER;

    public final int swigValue() {
        return swigValue;
    }

    public static Beverage swigToEnum(int swigValue) {
        Beverage[] swigValues = Beverage.class.getEnumConstants();
        if (swigValue < swigValues.length && swigValues[swigValue].swigValue == swigValue)
            return swigValues[swigValue];
        for (Beverage swigEnum : swigValues)
            if (swigEnum.swigValue == swigValue)
                return swigEnum;
        throw new IllegalArgumentException("No enum " + Beverage.class +
                                         " with value " + swigValue);
    }

    private Beverage() {
        this.swigValue = SwigNext.next++;
    }

    private Beverage(int swigValue) {
        this.swigValue = swigValue;
        SwigNext.next = swigValue+1;
    }

    private final int swigValue;

    private static class SwigNext {
        private static int next = 0;
    }
}
```

The enum items appear first. Like the typesafe enum pattern, the constructors are private. The constructors are required to handle C/C++ enums with initializers. The next variable is in the SwigNext inner class rather than in the enum class as static primitive variables cannot be modified from within enum constructors. Marshalling between Java enums and the C/C++ enum integer value is handled via the swigToEnum and swigValue methods. All the constructors and methods in the Java enum are required just to handle C/C++ enums with initializers. These needn't be generated if the enum being wrapped does not have any initializers and the [Simpler Java enums for enums without initializers](#) section describes how typemaps can be used to achieve this.

### 19.4.5.3 Type unsafe enum classes

The following example demonstrates type unsafe enums:

```
%include "enumtypeunsafe.swg"
%javaconst(1);
enum Beverage { ALE, LAGER=10, STOUT, PILSNER };
```

SWIG will generate the following simple class:

```
public final class Beverage {
    public final static int ALE = 0;
    public final static int LAGER = 10;
    public final static int STOUT = LAGER + 1;
    public final static int PILSNER = STOUT + 1;
}
```

## 19.5 Cross language polymorphism using directors (experimental)

Proxy classes provide a natural, object-oriented way to wrap C++ classes. as described earlier, each proxy instance has an associated C++ instance, and method calls from Java to the proxy are passed to the C++ instance transparently via C wrapper functions.

This arrangement is asymmetric in the sense that no corresponding mechanism exists to pass method calls down the inheritance chain from C++ to Java. In particular, if a C++ class has been extended in Java (by deriving from the proxy class), these classes will not be visible from C++ code. Virtual method calls from C++ are thus not able to access the lowest implementation in the inheritance chain.

SWIG can address this problem and make the relationship between C++ classes and proxy classes more symmetric. To achieve this goal, new classes called directors are introduced at the bottom of the C++ inheritance chain. The job of the directors is to route method calls correctly, either to C++ implementations higher in the inheritance chain or to Java implementations lower in the inheritance chain. The upshot is that C++ classes can be extended in Java and from C++ these extensions look exactly like native C++ classes. Neither C++ code nor Java code needs to know where a particular method is implemented: the combination of proxy classes, director classes, and C wrapper functions transparently takes care of all the cross-language method routing.

### 19.5.1 Enabling directors

The director feature is disabled by default. To use directors you must make two changes to the interface file. First, add the "directors" option to the %module directive, like this:

```
%module(directors="1") modulename
```

Without this option no director code will be generated. Second, you must use the %feature("director") directive to tell SWIG which classes and methods should get directors. The %feature directive can be applied globally, to specific classes, and to specific methods, like this:

```
// generate directors for all classes that have virtual methods
%feature("director");

// generate directors for all virtual methods in class Foo
%feature("director") Foo;

// generate a director for just Foo::bar()
%feature("director") Foo::bar;
```

You can use the %feature("nodirector") directive to turn off directors for specific classes or methods. So for example,

```
%feature("director") Foo;
%feature("nodirector") Foo::bar;
```

will generate directors for all virtual methods of class Foo except bar().

Directors can also be generated implicitly through inheritance. In the following, class Bar will get a director class that handles the methods one() and two() (but not three()):

```
%feature("director") Foo;
class Foo {
public:
    virtual void one();
```

```

    virtual void two();
};

class Bar: public Foo {
public:
    virtual void three();
};

```

## 19.5.2 Director classes

For each class that has directors enabled, SWIG generates a new class that derives from both the class in question and a special `Swig::Director` class. These new classes, referred to as director classes, can be loosely thought of as the C++ equivalent of the Java proxy classes. The director classes store a pointer to their underlying Java proxy classes.

For simplicity let's ignore the `Swig::Director` class and refer to the original C++ class as the director's base class. By default, a director class extends all virtual methods in the inheritance chain of its base class (see the preceding section for how to modify this behavior). Thus all virtual method calls, whether they originate in C++ or in Java via proxy classes, eventually end up in at the implementation in the director class. The job of the director methods is to route these method calls to the appropriate place in the inheritance chain. By "appropriate place" we mean the method that would have been called if the C++ base class and its Java derived classes were seamlessly integrated. That seamless integration is exactly what the director classes provide, transparently skipping over all the messy JNI glue code that binds the two languages together.

In reality, the "appropriate place" is one of only two possibilities: C++ or Java. Once this decision is made, the rest is fairly easy. If the correct implementation is in C++, then the lowest implementation of the method in the C++ inheritance chain is called explicitly. If the correct implementation is in Java, the Java API is used to call the method of the underlying Java object (after which the usual virtual method resolution in Java automatically finds the right implementation).

## 19.5.3 Overhead and code bloat

Enabling directors for a class will generate a new director method for every virtual method in the class' inheritance chain. This alone can generate a lot of code bloat for large hierarchies. Method arguments that require complex conversions to and from Java types can result in large director methods. For this reason it is recommended that directors are selectively enabled only for specific classes that are likely to be extended in Java and used in C++.

Although directors make it natural to mix native C++ objects with Java objects (as director objects), one should be aware of the obvious fact that method calls to Java objects from C++ will be much slower than calls to C++ objects. Additionally, compared to classes that do not use directors, the call routing in the director methods adds a small overhead. This situation can be optimized by selectively enabling director methods (using the `%feature` directive) for only those methods that are likely to be extended in Java.

## 19.5.4 Simple directors example

Consider the following SWIG interface file:

```

%module(directors="1") example;

%feature("director") DirectorBase;

class DirectorBase {
public:
    virtual ~DirectorBase() {}
    virtual void upcall_method() {}
};

void callup(DirectorBase *director) {
    director->upcall_method();
}

```

The following `directorDerived` Java class is derived from the Java proxy class `DirectorBase` and overrides `upcall_method()`. When C++ code invokes `upcall_method()`, the SWIG-generated C++ code redirects the call via JNI

to the Java `directorDerived` subclass. Naturally, the SWIG generated C++ code and the generated Java intermediate class marshal and convert arguments between C++ and Java when needed.

```
public class directorDerived extends DirectorBase {
    public directorDerived() {
    }

    public void upcall_method() {
        System.out.println("directorDerived::upcall_method() invoked.");
    }
}
```

Running the following Java code

```
directorDerived director = new directorDerived();
example.callup(director);
```

will result in the following being output:

```
directorDerived::upcall_method() invoked.
```

## 19.6 Common customization features

An earlier section presented the absolute basics of C/C++ wrapping. If you do nothing but feed SWIG a header file, you will get an interface that mimics the behavior described. However, sometimes this isn't enough to produce a nice module. Certain types of functionality might be missing or the interface to certain functions might be awkward. This section describes some common SWIG features that are used to improve the interface to existing C/C++ code.

### 19.6.1 C/C++ helper functions

Sometimes when you create a module, it is missing certain bits of functionality. For example, if you had a function like this

```
typedef struct Image {...};
void set_transform(Image *im, double m[4][4]);
```

it would be accessible from Java, but there may be no easy way to call it. The problem here is that a type wrapper class is generated for the two dimensional array parameter so there is no easy way to construct and manipulate a suitable `double [4][4]` value. To fix this, you can write some extra C helper functions. Just use the `%inline` directive. For example:

```
%inline %{
/* Note: double[4][4] is equivalent to a pointer to an array double (*)[4] */
double (*new_mat44())[4] {
    return (double (*)[4]) malloc(16*sizeof(double));
}
void free_mat44(double (*x)[4]) {
    free(x);
}
void mat44_set(double x[4][4], int i, int j, double v) {
    x[i][j] = v;
}
double mat44_get(double x[4][4], int i, int j) {
    return x[i][j];
}
%}
```

From Java, you could then write code like this:

```
Image im = new Image();
SWIGTYPE_p_a_4__double a = example.new_mat44();
example.mat44_set(a,0,0,1.0);
example.mat44_set(a,1,1,1.0);
```

```
example.mat44_set(a,2,2,1.0);
...
example.set_transform(im,a);
example.free_mat44(a);
```

Admittedly, this is not the most elegant looking approach. However, it works and it wasn't too hard to implement. It is possible to improve on this using Java code, typemaps, and other customization features as covered in later sections, but sometimes helper functions are a quick and easy solution to difficult cases.

## 19.6.2 Class extension with %extend

One of the more interesting features of SWIG is that it can extend structures and classes with new methods or constructors. Here is a simple example:

```
%module example
%{
#include "someheader.h"
%}

struct Vector {
    double x,y,z;
};

%extend Vector {
    char *toString() {
        static char tmp[1024];
        sprintf(tmp,"Vector(%g,%g,%g)", self->x,self->y,self->z);
        return tmp;
    }
    Vector(double x, double y, double z) {
        Vector *v = (Vector *) malloc(sizeof(Vector));
        v->x = x;
        v->y = y;
        v->z = z;
        return v;
    }
};
```

Now, in Java

```
Vector v = new Vector(2,3,4);
System.out.println(v);
```

will display

```
Vector(2,3,4)
```

%extend works with both C and C++ code. It does not modify the underlying object in any way—the extensions only show up in the Java interface.

## 19.6.3 Exception handling with %exception and %javaexception

If a C or C++ function throws an error, you may want to convert that error into a Java exception. To do this, you can use the %exception directive. The %exception directive simply lets you rewrite part of the generated wrapper code to include an error check. It is detailed in full in the [Exception handling with %exception](#) section.

In C, a function often indicates an error by returning a status code (a negative number or a NULL pointer perhaps). Here is a simple example of how you might handle that:

```
%exception malloc {
    $action
```



```

    if (!result) {
        jclass clazz = (*jenv)->FindClass(jenv, "java/lang/OutOfMemoryError");
        (*jenv)->ThrowNew(jenv, clazz, "Not enough memory");
        return $null;
    }
}
void *malloc(size_t nbytes);

```

In Java,

```
SWIGTYPE_p_void a = example.malloc(2000000000);
```

will produce a familiar looking Java exception:

```

Exception in thread "main" java.lang.OutOfMemoryError: Not enough memory
    at exampleJNI.malloc(Native Method)
    at example.malloc(example.java:16)
    at main.main(main.java:112)

```

If a library provides some kind of general error handling framework, you can also use that. For example:

```

%exception malloc {
    $action
    if (err_occurred()) {
        jclass clazz = (*jenv)->FindClass(jenv, "java/lang/OutOfMemoryError");
        (*jenv)->ThrowNew(jenv, clazz, "Not enough memory");
        return $null;
    }
}
void *malloc(size_t nbytes);

```

No declaration name is given to `%exception`, it is applied to all wrapper functions. The `$action` is a SWIG special variable and is replaced by the C/C++ function call being wrapped. The `return $null;` handles all native method return types, namely those that have a void return and those that do not. This is useful for typemaps that will be used in native method returning all return types. See the section on [Java special variables](#) for further explanation.

C++ exceptions are also easy to handle. We can catch the C++ exception and rethrow it as a Java exception like this:

```

%exception getitem {
    try {
        $action
    } catch (std::out_of_range &e) {
        jclass clazz = jenv->FindClass("java/lang/Exception");
        jenv->ThrowNew(clazz, "Range error");
        return $null;
    }
}

class FooClass {
public:
    FooClass *getitem(int index);        // Might throw std::out_of_range exception
    ...
};

```

In the example above, `java.lang.Exception` is a checked exception class and so ought to be declared in the `throws` clause of `getitem`. Classes can be specified for adding to the `throws` clause using `%javaexception(classes)` instead of `%exception`, where `classes` is a string containing one or more comma separated Java classes. The `%nojavaexception` feature is the equivalent to `%noexception` and clears previously declared exception handlers.

```

%javaexception("java.lang.Exception") getitem {
    try {
        $action
    } catch (std::out_of_range &e) {

```

```

    jclass clazz = jenv->FindClass("java/lang/Exception");
    jenv->ThrowNew(clazz, "Range error");
    return $null;
}

class FooClass {
public:
    FooClass *getitem(int index);      // Might throw std::out_of_range exception
    ...
};

```

The generated proxy method now generates a throws clause containing `java.lang.Exception`:

```

public class FooClass {
    ...
    public FooClass getitem(int index) throws java.lang.Exception { ... }
    ...
}

```

The examples above first use the C JNI calling syntax then the C++ JNI calling syntax. The C++ calling syntax will not compile as C and also visa versa. It is however possible to write JNI calls which will compile under both C and C++ and is covered in the [Typemaps for both C and C++ compilation](#) section.

The language-independent `exception.i` library file can also be used to raise exceptions. See the [SWIG Library](#) chapter. The typemap example [Handling C++ exception specifications as Java exceptions](#) provides further exception handling capabilities.

#### 19.6.4 Method access with %javamethodmodifiers

A Java feature called `%javamethodmodifiers` can be used to change the method modifiers from the default `public`. It applies to both module class methods and proxy class methods. For example:

```

%javamethodmodifiers protect_me() "protected";
void protect_me();

```

Will produce the method in the module class with protected access.

```

protected static void protect_me() {
    exampleJNI.protect_me();
}

```

## 19.7 Tips and techniques

Although SWIG is largely automatic, there are certain types of wrapping problems that require additional user input. Examples include dealing with output parameters, strings and arrays. This chapter discusses the common techniques for solving these problems.

### 19.7.1 Input and output parameters using primitive pointers and references

A common problem in some C programs is handling parameters passed as simple pointers or references. For example:

```

void add(int x, int y, int *result) {
    *result = x + y;
}

```

or perhaps

```

int sub(int *x, int *y) {
    return *x-*y;
}

```

The `typemaps.i` library file will help in these situations. For example:

```
%module example
#include "typemaps.i"

void add(int, int, int *OUTPUT);
int sub(int *INPUT, int *INPUT);
```

In Java, this allows you to pass simple values. For example:

```
int result = example.sub(7,4);
System.out.println("7 - 4 = " + result);
int[] sum = {0};
example.add(3,4,sum);
System.out.println("3 + 4 = " + sum[0]);
```

Which will display:

```
7 - 4 = 3
3 + 4 = 7
```

Notice how the `INPUT` parameters allow integer values to be passed instead of pointers and how the `OUTPUT` parameter will return the result in the first element of the integer array.

If you don't want to use the names `INPUT` or `OUTPUT`, use the `%apply` directive. For example:

```
%module example
#include "typemaps.i"

%apply int *OUTPUT { int *result };
%apply int *INPUT { int *x, int *y};

void add(int x, int y, int *result);
int sub(int *x, int *y);
```

If a function mutates one of its parameters like this,

```
void negate(int *x) {
    *x = -(*x);
}
```

you can use `INOUT` like this:

```
%include "typemaps.i"
...
void negate(int *INOUT);
```

In Java, the input parameter is the first element in a 1 element array and is replaced by the output of the function. For example:

```
int[] neg = {3};
example.negate(neg);
System.out.println("Negative of 3 = " + neg[0]);
```

And no prizes for guessing the output:

```
Negative of 3 = -3
```

These typemaps can also be applied to C++ references. The above examples would work the same if they had been defined using references instead of pointers. For example, the Java code to use the `negate` function would be the same if it were defined either as it is above:

```
void negate(int *INOUT);
```

or using a reference:

```
void negate(int &INOUT);
```

Note: Since most Java primitive types are immutable and are passed by value, it is not possible to perform in-place modification of a type passed as a parameter.

Be aware that the primary purpose of the `typemaps.i` file is to support primitive datatypes. Writing a function like this

```
void foo(Bar *OUTPUT);
```

will not have the intended effect since `typemaps.i` does not define an `OUTPUT` rule for `Bar`.

## 19.7.2 Simple pointers

If you must work with simple pointers such as `int *` or `double *` another approach to using `typemaps.i` is to use the `cpointer.i` pointer library file. For example:

```
%module example
#include "cpointer.i"

extern void add(int x, int y, int *result);
%pointer_functions(int, intp);
```

The `%pointer_functions(type, name)` macro generates five helper functions that can be used to create, destroy, copy, assign, and dereference a pointer. In this case, the functions are as follows:

```
int  *new_intp();
int  *copy_intp(int *x);
void  delete_intp(int *x);
void  intp_assign(int *x, int value);
int  intp_value(int *x);
```

In Java, you would use the functions like this:

```
SWIGTYPE_p_int intPtr = example.new_intp();
example.add(3,4,intPtr);
int result = example.intp_value(intPtr);
System.out.println("3 + 4 = " + result);
```

If you replace `%pointer_functions(int, intp)` by `%pointer_class(int, intp)`, the interface is more class-like.

```
intp intPtr = new intp();
example.add(3,4,intPtr.cast());
int result = intPtr.value();
System.out.println("3 + 4 = " + result);
```

See the [SWIG Library](#) chapter for further details.

## 19.7.3 Wrapping C arrays with Java arrays

SWIG can wrap arrays in a more natural Java manner than the default by using the `arrays_java.i` library file. Let's consider an example:

```
%include "arrays_java.i";
int array[4];
void populate(int x[]) {
    int i;
```

```
for (i=0; i
```

These one dimensional arrays can then be used as if they were Java arrays:

```
int[] array = new int[4];
example.populate(array);

System.out.print("array: ");
for (int i=0; i<array.length; i++)
    System.out.print(array[i] + " ");

example.setArray(array);

int[] global_array = example.getArray();

System.out.print("\nglobal_array: ");
for (int i=0; i<array.length; i++)
    System.out.print(global_array[i] + " ");
```

Java arrays are always passed by reference, so any changes a function makes to the array will be seen by the calling function. Here is the output after running this code:

```
array: 100 101 102 103
global_array: 100 101 102 103
```

Note that for assigning array variables the length of the C variable is used, so it is possible to use a Java array that is bigger than the C code will cope with. Only the number of elements in the C array will be used. However, if the Java array is not large enough then you are likely to get a segmentation fault or access violation, just like you would in C. When arrays are used in functions like `populate`, the size of the C array passed to the function is determined by the size of the Java array.

Please be aware that the typemaps in this library are not efficient as all the elements are copied from the Java array to a C array whenever the array is passed to and from JNI code. There is an alternative approach using the SWIG array library and this is covered in the next.

## 19.7.4 Unbounded C Arrays

Sometimes a C function expects an array to be passed as a pointer. For example,

```
int sumitems(int *first, int nitems) {
    int i, sum = 0;
    for (i = 0; i < nitems; i++) {
        sum += first[i];
    }
    return sum;
}
```

One of the ways to wrap this is to apply the Java array typemaps that come in the `arrays_java.i` library file:

```
%include "arrays_java.i"
%apply int[ANY] {int *};
```

The ANY size will ensure the typemap is applied to arrays of all sizes. You could narrow the typemap matching rules by specifying a particular array size. Now you can use a pure Java array and pass it to the C code:

```
int[] array = new int[10000000];           // Array of 10-million integers
for (int i=0; i<array.length; i++) {      // Set some values
    array[i] = i;
}
int sum = example.sumitems(array,10000);
System.out.println("Sum = " + sum);
```

and the sum would be displayed:

```
Sum = 49995000
```

This approach is probably the most natural way to use arrays. However, it suffers from performance problems when using large arrays as a lot of copying of the elements occurs in transferring the array from the Java world to the C++ world. An alternative approach to using Java arrays for C arrays is to use an alternative SWIG library file `carrays.i`. This approach can be more efficient for large arrays as the array is accessed one element at a time. For example:

```
%include "carrays.i"
%array_functions(int, intArray);
```

The `%array_functions(type, name)` macro generates four helper functions that can be used to create and destroy arrays and operate on elements. In this case, the functions are as follows:

```
int *new_intArray(int nelements);
void delete_intArray(int *x);
int intArray_getitem(int *x, int index);
void intArray_setitem(int *x, int index, int value);
```

In Java, you would use the functions like this:

```
SWIGTYPE_p_int array = example.new_intArray(10000000); // Array of 10-million integers
for (int i=0; i
```

If you replace `%array_functions(int, intp)` by `%array_class(int, intp)`, the interface is more class-like and a couple more helper functions are available for casting between the array and the type wrapper class.

```
%include "carrays.i"
%array_class(int, intArray);
```

The `%array_class(type, name)` macro creates wrappers for an unbounded array object that can be passed around as a simple pointer like `int *` or `double *`. For instance, you will be able to do this in Java:

```
intArray array = new intArray(10000000); // Array of 10-million integers
for (int i=0; i
```

The array "object" created by `%array_class()` does not encapsulate pointers inside a special array object. In fact, there is no bounds checking or safety of any kind (just like in C). Because of this, the arrays created by this library are extremely low-level indeed. You can't iterate over them nor can you even query their length. In fact, any valid memory address can be accessed if you want (negative indices, indices beyond the end of the array, etc.). Needless to say, this approach is not going to suit all applications. On the other hand, this low-level approach is extremely efficient and well suited for applications in which you need to create buffers, package binary data, etc.

## 19.8 Java typemaps

This section describes how you can modify SWIG's default wrapping behavior for various C/C++ datatypes using the `%typemap` directive. You are advised to be familiar with the the material in the "[Typemaps](#)" chapter. While not absolutely essential knowledge, this section assumes some familiarity with the Java Native Interface (JNI). JNI documentation can be consulted either online at [Sun's Java web site](#) or from a good JNI book. The following two books are recommended:

- Title: 'Essential JNI: Java Native Interface.' Author: Rob Gordon. Publisher: Prentice Hall. ISBN: 0-13-679895-0.
- Title: 'The Java Native Interface: Programmer's Guide and Specification.' Author: Sheng Liang. Publisher: Addison-Wesley. ISBN: 0-201-32577-2.

Before proceeding, it should be stressed that typemaps are not a required part of using SWIG---the default wrapping behavior is enough in most cases. Typemaps are only used if you want to change some aspect of the generated code.

## 19.8.1 Default primitive type mappings

The following table lists the default type mapping from Java to C/C++.

C/C++ type	Java type	JNI type
bool const bool &	boolean	jboolean
char const char &	char	jchar
signed char const signed char &	byte	jbyte
unsigned char const unsigned char &	short	jshort
short const short &	short	jshort
unsigned short const unsigned short &	int	jint
int const int &	int	jint
unsigned int const unsigned int &	long	jlong
long const long &	int	jint
unsigned long const unsigned long &	long	jlong
long long const long long &	long	jlong
unsigned long long const unsigned long long &	java.math.BigInteger	jobject
float const float &	float	jfloat
double const double &	double	jdouble
char * char []	String	jstring

Note that SWIG wraps the C char type as a character. Pointers and arrays of this type are wrapped as strings. The `signed char` type can be used if you want to treat `char` as a signed number rather than a character. Also note that all `const` references to primitive types are treated as if they are passed by value.

Given the following C function:

```
void func(unsigned short a, char *b, const long &c, unsigned long long d);
```

The module class method would be:

```
public static void func(int a, String b, int c, java.math.BigInteger d) {...}
```

The intermediary JNI class would use the same types:

```
public final static native void func(int jarg1, String jarg2, int jarg3,
                                     java.math.BigInteger jarg4);
```

and the JNI function would look like this:

```
JNIEXPORT void JNICALL Java_exampleJNI_func(JNIEnv *jenv, jclass jcls,
                                             jint jarg1, jstring jarg2, jint jarg3, jobject jarg4) {...}
```

The mappings for C `int` and C `long` are appropriate for 32 bit applications which are used in the 32 bit JVMs. There is no perfect mapping between Java and C as Java doesn't support all the unsigned C data types. However, the mappings allow the full range of values for each C type from Java.

## 19.8.2 Sixty four bit JVMs

If you are using a 64 bit JVM you may have to override the C `long`, but probably not C `int` default mappings. Mappings will be system dependent, for example `long` will need remapping on Unix LP64 systems (`long`, pointer 64 bits, `int` 32 bits), but not on Microsoft 64 bit Windows which will be using a P64 IL32 (pointer 64 bits and `int`, `long` 32 bits) model. This may be automated in a future version of SWIG. Note that the Java write once run anywhere philosophy holds true for all pure Java code when moving to a 64 bit JVM. Unfortunately it won't of course hold true for JNI code.

## 19.8.3 What is a typemap?

A typemap is nothing more than a code generation rule that is attached to a specific C datatype. For example, to convert integers from Java to C, you might define a typemap like this:

```
%module example

%typemap(in) int {
    $1 = $input;
    printf("Received an integer : %d\n", $1);
}

extern int fact(int nonnegative);
```

Typemaps are always associated with some specific aspect of code generation. In this case, the "in" method refers to the conversion of input arguments to C/C++. The datatype `int` is the datatype to which the typemap will be applied. The supplied C code is used to convert values. In this code a number of special variables prefaced by a `$` are used. The `$1` variable is a placeholder for a local variable of type `int`. The `$input` variable contains the Java data, the JNI `jint` in this case.

When this example is compiled into a Java module, it can be used as follows:

```
System.out.println(example.fact(6));
```

and the output will be:

```
Received an integer : 6
720
```

In this example, the typemap is applied to all occurrences of the `int` datatype. You can refine this by supplying an optional parameter name. For example:

```
%module example

%typemap(in) int nonnegative {
    $1 = $input;
    printf("Received an integer : %d\n", $1);
}

extern int fact(int nonnegative);
```

In this case, the typemap code is only attached to arguments that exactly match `int nonnegative`.



The application of a typemap to specific datatypes and argument names involves more than simple text-matching—typemaps are fully integrated into the SWIG C++ type-system. When you define a typemap for `int`, that typemap applies to `int` and qualified variations such as `const int`. In addition, the typemap system follows `typedef` declarations. For example:

```
%typemap(in) int nonnegative {
    $1 = $input;
    printf("Received an integer : %d\n", $1);
}
typedef int Integer;
extern int fact(Integer nonnegative);    // Above typemap is applied
```

However, the matching of `typedef` only occurs in one direction. If you defined a typemap for `Integer`, it is not applied to arguments of type `int`.

Typemaps can also be defined for groups of consecutive arguments. For example:

```
%typemap(in) (char *str, int len) {
    ...
};

int count(char c, char *str, int len);
```

When a multi-argument typemap is defined, the arguments are always handled as a single Java parameter. This allows the function to be used like this (notice how the length parameter is omitted):

```
int c = example.count('e', "Hello World");
```

## 19.8.4 Typemaps for mapping C/C++ types to Java types

The typemaps available to the Java module include the common typemaps listed in the main typemaps section. There are a number of additional typemaps which are necessary for using SWIG with Java. The most important of these implement the mapping of C/C++ types to Java types:

Typemap	Description
jni	JNI C types. These provide the default mapping of types from C/C++ to JNI for use in the JNI (C/C++) code.
jtype	Java intermediary types. These provide the default mapping of types from C/C++ to Java for use in the native functions in the intermediary JNI class. The type must be the equivalent Java type for the JNI C type specified in the "jni" typemap.
jstype	Java types. These provide the default mapping of types from C/C++ to Java for use in the Java module class, proxy classes and type wrapper classes.
javain	Conversion from jstype to jtype. These are Java code typemaps which transform the type used in the Java module class, proxy classes and type wrapper classes (as specified in the "jstype" typemap) to the type used in the Java intermediary JNI class (as specified in the "jtype" typemap). In other words the typemap provides the conversion to the native method call parameter types.
javaout	Conversion from jtype to jstype. These are Java code typemaps which transform the type used in the Java intermediary JNI class (as specified in the "jtype" typemap) to the Java type used in the Java module class, proxy classes and type wrapper classes (as specified in the "jstype" typemap). In other words the typemap provides the conversion from the native method call return type.
javadirectorin	Conversion from jtype to jstype for director methods. These are Java code typemaps which transform the type used in the Java intermediary JNI class (as specified in the "jtype" typemap) to the Java type used in the Java module class, proxy classes and type wrapper classes (as specified in the "jstype" typemap). This typemap provides the conversion for the parameters in the director methods when calling up from C++ to Java. See <a href="#">Director typemaps</a> .
javadirectorout	

	Conversion from jstype to jtype for director methods. These are Java code typemaps which transform the type used in the Java module class, proxy classes and type wrapper classes (as specified in the "jstype" typemap) to the type used in the Java intermediary JNI class (as specified in the "jtype" typemap). This typemap provides the conversion for the return type in the director methods when returning from the C++ to Java upcall. See <a href="#">Director typemaps</a> .
directorin	Conversion from C++ type to jni type for director methods. These are C++ typemaps which converts the parameters used in the C++ director method to the appropriate JNI intermediary type. The conversion is done in JNI code prior to calling the Java function from the JNI code. See <a href="#">Director typemaps</a> .

If you are writing your own typemaps to handle a particular type, you will normally have to write a collection of them. The default typemaps are in "java.swg" and so might be a good place for finding typemaps to base any new ones on.

The "jni", "jtype" and "jstype" typemaps are usually defined together to handle the Java to C/C++ type mapping. An "in" typemap should be accompanied by a "javain" typemap and likewise an "out" typemap by a "javaout" typemap. If an "in" typemap is written, a "freearg" and "argout" typemap may also need to be written as some types have a default "freearg" and/or "argout" typemap which may need overriding. The "freearg" typemap sometimes releases memory allocated by the "in" typemap. The "argout" typemap sometimes sets values in function parameters which are passed by reference in Java.

The default code generated by SWIG for the Java module comes from the typemaps in the "java.swg" library file which implements the [Default primitive type mappings](#) covered earlier. There are other type mapping typemaps in the Java library. These are listed below:

C Type	Typemap	File	Kind	Java Type	Function
primitive pointers and references	INPUT	typemaps.i	input	Java basic types	Allows values to be used for C functions taking pointers for data input.
primitive pointers and references	OUTPUT	typemaps.i	output	Java basic type arrays	Allows values held within an array to be used for C functions taking pointers for data output.
primitive pointers and references	INOUT	typemaps.i	input output	Java basic type arrays	Allows values held within an array to be used for C functions taking pointers for data input and output.
string wstring	[unnamed]	std_string.i	input output	String	Use for std::string mapping to Java String.
arrays of primitive types	[unnamed]	arrays_java.i	input output	arrays of primitive Java types	Use for mapping C arrays to Java arrays.
arrays of classes/structs/unions	JAVA_ARRAYSOFCLASSES macro	arrays_java.i	input output	arrays of proxy classes	Use for mapping C arrays to Java arrays.
arrays of enums	ARRAYSOFENUMS	arrays_java.i	input output	int[]	Use for mapping C arrays to Java arrays (typeunsafe and simple enum wrapping approaches only).
char *	BYTE	various.i	input	byte[]	Java byte array is converted to char array
char **	STRING_ARRAY	various.i	input output	String[]	Use for mapping NULL terminated arrays of C strings to Java String arrays

## 19.8.5 Java typemap attributes

There is an additional typemap attribute that the Java module supports. This is the 'throws' attribute. The throws attribute is optional and specified after the typemap name and contains one or more comma separated classes for adding to the throws clause for any methods that use that typemap. It is analogous to the [%javaexception](#) feature's throws attribute.

```
%typemap(typemapname, throws="ExceptionClass1, ExceptionClass2") type { ... }
```

The attribute is necessary for supporting Java checked exceptions and can be added to just about any typemap. The list of typemaps include all the C/C++ (JNI) typemaps in the "[Typemaps](#)" chapter and the Java specific typemaps listed in [the previous section](#), barring the "jni", "jtype" and "jstype" typemaps as they could never contain code to throw an exception.

The throws clause is generated for the proxy method as well as the JNI method in the JNI intermediary class. If a method uses more than one typemap and each of those typemaps have classes specified in the throws clause, the union of the exception classes is added to the throws clause ensuring there are no duplicate classes. See the [NaN exception example](#) for further usage.

## 19.8.6 Java special variables

The standard SWIG special variables are available for use within typemaps as described in the [Typemaps documentation](#), for example \$1, \$input, \$result etc.

The Java module uses a few additional special variables:

### **\$javaclassname**

\$javaclassname is similar to \$1\_type. It expands to the class name for use in Java. When wrapping a union, struct or class, it expands to the Java proxy class name. Otherwise it expands to the type wrapper class name. For example, \$javaclassname is replaced by Foo when the wrapping a struct Foo or struct Foo \* and SWIGTYPE\_p\_unsigned\_short is used for unsigned short \*.

### **\$null**

Used in input typemaps to return early from JNI functions that have either void or a non-void return type. Example:

```
%typemap(check) int * %{
    if (error) {
        SWIG_exception(SWIG_IndexError, "Array element error");
        return $null;
    }
}%
```

If the typemap gets put into a function with void as return, \$null will expand to nothing:

```
JNIEXPORT void JNICALL Java_jnifn(...) {
    if (error) {
        SWIG_exception(SWIG_IndexError, "Array element error");
        return ;
    }
    ...
}
```

otherwise \$null expands to *NULL*

```
JNIEXPORT jobject JNICALL Java_jnifn(...) {
    if (error) {
        SWIG_exception(SWIG_IndexError, "Array element error");
        return NULL;
    }
    ...
}
```

**\$javainput, \$jnicall and \$owner**

The \$javainput special variable is used in "javain" typemaps and \$jnicall and \$owner are used in "javaout" typemaps. \$jnicall is analogous to \$action in %exception. It is replaced by the call to the native method in the intermediary JNI class. \$owner is replaced by either true if %newobject has been used, otherwise false. \$javainput is analogous to the \$input special variable. It is replaced by the parameter name.

Here is an example:

```
%typemap(javain) Class "Class.getCPtr($javainput)"
%typemap(javain) unsigned short "$javainput"
%typemap(javaout) Class * {
    return new Class($jnicall, $owner);
}

%inline %{
    class Class {...};
    Class * bar(Class cls, unsigned short ush) { return new Class(); };
}%
```

The generated proxy code is then:

```
public static Class bar(Class cls, int ush) {
    return new Class(exampleJNI.bar(Class.getCPtr(cls), ush), false);
}
```

Here \$javainput has been replaced by cls and ush. \$jnicall has been replaced by the native method call, exampleJNI.bar(...) and \$owner has been replaced by false. If %newobject is used by adding the following at the beginning of our example:

```
%newobject bar(Class cls, unsigned short ush);
```

The generated code constructs the return type using true indicating the proxy class Class is responsible for destroying the C++ memory allocated for it in bar:

```
public static Class bar(Class cls, int ush) {
    return new Class(exampleJNI.bar(Class.getCPtr(cls), ush), true);
}
```

**\$static**

This special variable expands to either *static* or nothing depending on whether the class is an inner Java class or not. It is used in the "javamodifiers" typemap so that global classes can be wrapped as Java proxy classes and nested C++ classes/enums can be wrapped with the Java equivalent, that is, static inner proxy classes.

**\$jniinput, \$javacall and \$packagepath**

These special variables are used in the directors typemaps. See [Director specific typemaps](#) for details.

**\$module**

This special variable expands to the module name, as specified by %module or the -module commandline option. Useful for constructing the intermediary classname, which is just \$moduleJNI.

**19.8.7 Typemaps for both C and C++ compilation**

JNI calls must be written differently depending on whether the code is being compiled as C or C++. For example C compilation requires the pointer to a function pointer struct member syntax like

```
const jclass clazz = (*jenv)->FindClass(jenv, "java/lang/String");
```

whereas C++ code compilation of the same function call is a member function call using a class pointer like

```
const jclass clazz = jenv->FindClass("java/lang/String");
```

To enable typemaps to be used for either C or C++ compilation, a set of JCALLx macros have been defined in Lib/java/javahead.swg, where x is the number of arguments in the C++ version of the JNI call. The above JNI calls would be written in a typemap like this

```
const jclass clazz = JCALL1(FindClass, jenv, "java/lang/String");
```

Note that the SWIG preprocessor expands these into the appropriate C or C++ JNI calling convention. The C calling convention is emitted by default and the C++ calling convention is emitted when using the `-c++` SWIG commandline option. If you do not intend your code to be targeting both C and C++ then your typemaps can use the appropriate JNI calling convention and need not use the JCALLx macros.

### 19.8.8 Java code typemaps

Most of SWIG's typemaps are used for the generation of C/C++ code. The typemaps in this section are used solely for the generation of Java code. Elements of proxy classes and type wrapper classes come from the following typemaps (the defaults).

```
%typemap( javabase )
```

base (extends) for Java class: empty default

```
%typemap( javabody )
```

the essential support body for proxy classes (proxy base classes only), typewrapper classes and enum classes. Default contains extra constructors, memory ownership control member variables (swigCMemOwn, swigCPtr), the getCPtr method etc.

```
%typemap( javabody_derived )
```

the essential support body for proxy classes (derived classes only). Same as "javabody" typemap, but only used for proxy derived classes.

```
%typemap( javaclassmodifiers )
```

class modifiers for the Java class: default is "public class"

```
%typemap( javacode )
```

Java code is copied verbatim to the Java class: empty default

```
%typemap( javadestruct, methodname="delete" )
```

destructor wrapper – the `delete()` method (proxy classes only), used for all proxy classes except those which have a base class : default calls C++ destructor (or frees C memory) and resets swigCPtr and swigCMemOwn flags

Note that the `delete()` method name is configurable and is specified by the `methodname` attribute.

```
%typemap( javadestruct_derived, methodname="delete" )
```

destructor wrapper – the `delete()` method (proxy classes only), same as "javadestruct" but only used for derived proxy classes : default calls C++ destructor (or frees C memory) and resets swigCPtr and swigCMemOwn flags

Note that the `delete()` method name is configurable and is specified by the `methodname` attribute.

```
%typemap( javaimports )
```

import statements for Java class: empty default

```
%typemap( javainterfaces )
```

interfaces (extends) for Java class: empty default

```
%typemap( javafinalize )
```

the `finalize()` method (proxy classes only): default calls the `delete()` method

**Compatibility Note:** In SWIG-1.3.21 and earlier releases, typemaps called "javagetcptr" and "javaptrconstructormodifiers" were available. These are deprecated and the "javabody" typemap can be used instead.

In summary the contents of the typemaps make up a proxy class like this:

```
[ javaimports typemap ]
[ javaclassmodifiers typemap ] javaclassname extends [ javabase typemap ]
                                   implements [ javainterfaces typemap ] {
    [ javabody or javabody_derived typemap ]
    [ javafinalize typemap ]
    public void delete() [ javadestruct OR javadestruct_derived typemap ]
    [ javacode typemap ]
    ... proxy functions ...
}
```

Note the `delete()` methodname is configurable, see "javadestruct" and "javadestruct\_derived" typemaps above.

The type wrapper class is similar in construction:

```
[ javaimports typemap ]
[ javaclassmodifiers typemap ] javaclassname extends [ javabase typemap ]
                                   implements [ javainterfaces typemap ] {
    [ javabody typemap ]
    [ javacode typemap ]
}
```

The enum class is also similar in construction:

```
[ javaimports typemap ]
[ javaclassmodifiers typemap ] javaclassname extends [ javabase typemap ]
                                   implements [ javainterfaces typemap ] {
    ... Enum values ...
    [ javabody typemap ]
    [ javacode typemap ]
}
```

The "javaimports" typemap is ignored if the enum class is wrapped by an inner Java class, that is when wrapping an enum declared within a C++ class.

The defaults can be overridden to tailor these classes. Here is an example which will change the `getCPtr` method and constructor from the default protected access to public access. This has a practical application if you are invoking SWIG more than once and generating the wrapped classes into different packages in each invocation. If the classes in one package are using the classes in another package, then these methods need to be public.

```
%typemap(javabody) SWIGTYPE %{
    private long swigCPtr;
    protected boolean swigCMemOwn;

    public $javaclassname(long cPtr, boolean cMemoryOwn) {
```

```

        swigCMemOwn = cMemoryOwn;
        swigCPtr = cPtr;
    }

    public static long getCPtr($javaclassname obj) {
        return (obj == null) ? 0 : obj.swigCPtr;
    }
%}

```

The typemap code is the same that is in "java.swg", barring the two method modifiers. Note that SWIGTYPE will target all proxy classes, but not the type wrapper classes. Also the above typemap is only used for proxy classes that are potential base classes. To target proxy classes that are derived from a wrapped class as well, the "javabody\_derived" typemap should also be overridden.

For the typemap to be used in all type wrapper classes, all the different types that type wrapper classes could be used for should be targeted:

```

%typemap(javabody) SWIGTYPE *, SWIGTYPE &, SWIGTYPE [], SWIGTYPE (CLASS::*) %{
    private long swigCPtr;

    public $javaclassname(long cPtr, boolean bFutureUse) {
        swigCPtr = cPtr;
    }

    protected $javaclassname() {
        swigCPtr = 0;
    }

    public static long getCPtr($javaclassname obj) {
        return (obj == null) ? 0 : obj.swigCPtr;
    }
%}

```

Again this is the same that is in "java.swg", barring the method modifier for getCPtr.

### 19.8.9 Director specific typemaps

The Java directors feature requires the "javadirectorin", "javadirectorout" and the "directorin" typemaps in order to work properly. The "javapackage" typemap is an optional typemap used to identify the Java package path for individual SWIG generated proxy classes.

```
%typemap(directorin)
```

The "directorin" typemap is used for converting arguments in the C++ director class to the appropriate JNI type before the upcall to Java. This typemap also specifies the JNI field descriptor for the type in the "descriptor" attribute. For example, integers are converted as follows:

```
%typemap(directorin,descriptor="I") int "$input = (jint) $1;"
```

\$input is the SWIG name of the JNI temporary variable passed to Java in the upcall. The descriptor="I" will put an I into the JNI field descriptor that identifies the Java method that will be called from C++. For more about JNI field descriptors and their importance, refer to the [JNI documentation mentioned earlier](#). A typemap for C character strings is:

```
%typemap(directorin,descriptor="Ljava/lang/String;") char *
%{ $input = jenv->NewStringUTF($1); %}
```

User-defined types have the default "descriptor" attribute "L\$packagepath/\$javaclassname;" where \$packagepath is the package name passed from the SWIG command line and \$javaclassname is the Java proxy class' name. If the -package commandline option is not used to specify the package, then

'\$packagepath/' will be removed from the resulting output JNI field descriptor. **Do not forget the terminating ';' for JNI field descriptors starting with 'L'.** If the ';' is left out, Java will generate a "method not found" runtime error.

```
%typemap( javadirectorin)
```

Conversion from jtype to jstype for director methods. These are Java code typemaps which transform the type used in the Java intermediary JNI class (as specified in the "jtype" typemap) to the Java type used in the Java module class, proxy classes and type wrapper classes (as specified in the "jstype" typemap). This typemap provides the conversion for the parameters in the director methods when calling up from C++ to Java. For primitive types, this typemap is usually specified as:

```
%typemap( javadirectorin) int "$jniinput"
```

The \$jniinput special variable is analogous to \$javainput special variable. It is replaced by the input parameter name.

```
%typemap( javadirectorout)
```

Conversion from jstype to jtype for director methods. These are Java code typemaps which transform the type used in the Java module class, proxy classes and type wrapper classes (as specified in the "jstype" typemap) to the type used in the Java intermediary JNI class (as specified in the "jtype" typemap). This typemap provides the conversion for the return type in the director methods when returning from the C++ to Java upcall. For primitive types, this typemap is usually specified as:

```
%typemap( javadirectorout) int "$javacall"
```

The \$javacall special variable is analogous to the \$jnical1 special variable. It is replaced by the call to the target Java method. The target method is the method in the Java proxy class which overrides the virtual C++ method in the C++ base class.

```
%typemap( javapackage)
```

The "javapackage" typemap is optional; it serves to identify a class's Java package. This typemap should be used in conjunction with classes that are defined outside of the current SWIG interface file. For example:

```
// class Foo is handled in a different interface file:
%import "Foo.i"

%feature("director") Example;

%inline {
    class Bar { };

    class Example {
    public:
        virtual ~Example();
        void    ping(Foo *arg1, Bar *arg2);
    };
}
```

Assume that the Foo class is part of the Java package *wombat.foo* but the above interface file is part of the Java package *wombat.example*. Without the "javapackage" typemap, SWIG will assume that the Foo class belongs to *wombat.example* class. The corrected interface file looks like:

```
// class Foo is handled in a different interface file:
%import "Foo.i"
%typemap("javapackage") Foo "wombat.foo";
%feature("director") Example;
```



```
%inline {
    class Bar { };

    class Example {
    public:
        virtual ~Example();
        void    ping(Foo *arg1, Bar *arg2);
    };
}
```

Practically speaking, you should create a separate SWIG interface file, which is %import-ed into each SWIG interface file, when you have multiple Java packages:

```
%typemap("javapackage") SWIGTYPE, SWIGTYPE *, SWIGTYPE &
                                "package.for.most.classes";

%typemap("javapackage") Package_2_class_one "package.for.other.classes";
%typemap("javapackage") Package_3_class_two "package.for.another.set";
/* etc */
```

The basic strategy here is to provide a default package typemap for the majority of the classes, only providing "javapackage" typemaps for the exceptions.

## 19.9 Typemap Examples

This section includes a few examples of typemaps. For more examples, you might look at the files "java.swg" and "typemaps.i" in the SWIG library.

### 19.9.1 Simpler Java enums for enums without initializers

The default [Proper Java enums](#) approach to wrapping enums is somewhat verbose. This is to handle all possible C/C++ enums, in particular enums with initializers. The generated code can be simplified if the enum being wrapped does not have any initializers. The following shows how to remove the support methods that are generated by default and instead use the methods in the Java enum base class `java.lang.Enum` and `java.lang.Class` for marshalling enums between C/C++ and Java. The type used for the typemaps below is `enum SWIGTYPE` which is the default type used for all enums. The "enums.swg" file should be examined in order to see the original overridden versions of the typemaps.

```
%include "enums.swg"

%typemap(javain) enum SWIGTYPE "$javainput.ordinal()"
%typemap(javaout) enum SWIGTYPE {
    return $javaclassname.class.getEnumConstants()[ $jnicall ];
}
%typemap(javabody) enum SWIGTYPE ""

%inline %{
    enum HairType { blonde, ginger, brunette };
    void setHair(HairType h);
    HairType getHair();
}%
```

SWIG will generate the following Java enum, which is somewhat simpler than the default:

```
public enum HairType {
    blonde,
    ginger,
    brunette;
}
```

and the two Java proxy methods will be:

```

public static void setHair(HairType h) {
    exampleJNI.setHair(h.ordinal());
}

public static HairType getHair() {
    return HairType.class.getEnumConstants()[exampleJNI.getHair()];
}

```

For marshalling Java enums to C/C++ enums, the `ordinal` method is used to convert the Java enum into an integer value for passing to the JNI layer, see the "javain" typemap. For marshalling C/C++ enums to Java enums, the C/C++ enum value is cast to an integer in the C/C++ typemaps (not shown). This integer value is then used to index into the array of enum constants that the Java language provides. See the `getEnumConstants` method in the "javaout" typemap.

These typemaps can often be used as the default for wrapping enums as in many cases there won't be any enum initializers. In fact a good strategy is to always use these typemaps and to specifically handle enums with initializers using `%apply`. This would be done by using the original versions of these typemaps in "enums.swg" under another typemap name for applying using `%apply`.

## 19.9.2 Handling C++ exception specifications as Java exceptions

This example demonstrates various ways in which C++ exceptions can be tailored and converted into Java exceptions. Let's consider a simple file class `SimpleFile` and an exception class `FileException` which it may throw on error:

```

#include "std_string.i" // for std::string typemaps
#include <string>

class FileException {
    std::string message;
public:
    FileException(const std::string& msg) : message(msg) {}
    std::string what() {
        return message;
    }
};

class SimpleFile {
    std::string filename;
public:
    SimpleFile(const std::string& filename) : filename(filename) {}
    void open() throw(FileException) {
        ...
    }
};

```

As the `open` method has a C++ exception specification, SWIG will parse this and know that the method can throw an exception. The "[throws](#)" typemap is then used when SWIG encounters an exception specification. The default generic "throws" typemap looks like this:

```

%typemap(throws) SWIGTYPE, SWIGTYPE &, SWIGTYPE *, SWIGTYPE [ANY] %{
    SWIG_JavaThrowException(jenv, SWIG_JavaRuntimeException,
                           "C++ $1_type exception thrown");
    return $null;
}%

```

Basically SWIG will generate a C++ try catch block and the body of the "throws" typemap constitutes the catch block. The above typemap calls a SWIG supplied method which throws a `java.lang.RuntimeException`. This exception class is a runtime exception and therefore not a checked exception. If, however, we wanted to throw a checked exception, say `java.io.IOException`, then we could use the following typemap:

```

%typemap(throws, throws="java.io.IOException") FileException {
    jclass excep = jenv->FindClass("java/io/IOException");
    if (excep)
        jenv->ThrowNew(excep, $1.what().c_str());
}

```

```
    return $null;
}
```

Note that this typemap uses the 'throws' [typemap attribute](#) to ensure a throws clause is generated. The generated proxy method then specifies the checked exception by containing `java.io.IOException` in the throws clause:

```
public class SimpleFile {
    ...
    public void open() throws java.io.IOException { ... }
}
```

Lastly, if you don't want to map your C++ exception into one of the standard Java exceptions, the C++ class can be wrapped and turned into a custom Java exception class. If we go back to our example, the first thing we must do is get SWIG to wrap `FileException` and ensure that it derives from `java.lang.Exception`. Additionally, we might want to override the `java.lang.Exception.getMessage()` method. The typemaps to use then are as follows:

```
%typemap(javabase) FileException "java.lang.Exception";
%typemap(javacode) FileException %{
    public String getMessage() {
        return what();
    }
}%}
```

This generates:

```
public class FileException extends java.lang.Exception {
    ...
    public String getMessage() {
        return what();
    }

    public FileException(String msg) { ... }

    public String what() {
        return exampleJNI.FileException_what(swigCPtr);
    }
}
```

We could alternatively have used `%rename` to rename `what()` into `getMessage()`.

### 19.9.3 NaN Exception – exception handling for a particular type

A Java exception can be thrown from any Java or JNI code. Therefore, as most typemaps contain either Java or JNI code, just about any typemap could throw an exception. The following example demonstrates exception handling on a type by type basis by checking for 'Not a number' (NaN) whenever a parameter of type `float` is wrapped.

Consider the following C++ code:

```
bool calculate(float first, float second);
```

To validate every `float` being passed to C++, we could precede the code being wrapped by the following typemap which throws a runtime exception whenever the `float` is 'Not a Number':

```
%module example
%typemap(javain) float "$module.CheckForNaN($javainput)"
%pragma(java) modulecode=%{
    /** Simply returns the input value unless it is not a number,
        whereupon an exception is thrown. */
    static protected float CheckForNaN(float num) {
        if (Float.isNaN(num))
            throw new RuntimeException("Not a number");
        return num;
    }
}
```

```
    }
    %}
```

Note that the `CheckForNaN` support method has been added to the module class using the `modulecode` pragma. The following shows the generated code of interest:

```
public class example {
    ...

    /** Simply returns the input value unless it is not a number,
     *   whereupon an exception is thrown. */
    static protected float CheckForNaN(float num) {
        if (Float.isNaN(num))
            throw new RuntimeException("Not a number");
        return num;
    }

    public static boolean calculate(float first, float second) {
        return exampleJNI.calculate(example.CheckForNaN(first), example.CheckForNaN(second));
    }
}
```

Note that the "javain" typemap is used for every occurrence of a float being used as an input. Of course, we could have targetted the typemap at a particular parameter by using `float first`, say, instead of just `float`. If we decide that what we actually want is a checked exception instead of a runtime exception, we can change this easily enough. The proxy method that uses `float` as an input, must then add the exception class to the `throws` clause. SWIG can handle this as it supports the 'throws' [typemap attribute](#) for specifying classes for the `throws` clause. Thus we can modify the pragma and the typemap for the `throws` clause:

```
%typemap(javain, throws="java.lang.Exception") float "$module.CheckForNaN($javainput)"
%pragma(java) modulecode=%{
    /** Simply returns the input value unless it is not a number,
     *   whereupon an exception is thrown. */
    static protected float CheckForNaN(float num) throws java.lang.Exception {
        if (Float.isNaN(num))
            throw new RuntimeException("Not a number");
        return num;
    }
}%}
```

The `calculate` method now has a `throws` clause and even though the typemap is used twice for both `float first` and `float second`, the `throws` clause contains a single instance of `java.lang.Exception`:

```
public class example {
    ...

    /** Simply returns the input value unless it is not a number,
     *   whereupon an exception is thrown. */
    static protected float CheckForNaN(float num) throws java.lang.Exception {
        if (Float.isNaN(num))
            throw new RuntimeException("Not a number");
        return num;
    }

    public static boolean calculate(float first, float second) throws java.lang.Exception {
        return exampleJNI.calculate(example.CheckForNaN(first), example.CheckForNaN(second));
    }
}
```

If we were a martyr to the JNI cause, we could replace the succinct code within the "javain" typemap with a few pages of JNI code. If we had, we would have put it in the "in" typemap which, like all JNI and Java typemaps, also supports the 'throws' attribute.

## 19.9.4 Converting Java String arrays to char \*\*

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Java String array to be used as a char \*\* object.

```
%module example

/* This tells SWIG to treat char ** as a special case when used as a parameter
   in a function call */
%typemap(in) char ** (jint size) {
    jint i = 0;
    size = (*jenv)->GetArrayLength(jenv, $input);
    $1 = (char **) malloc((size+1)*sizeof(char *));
    /* make a copy of each string */
    for (i = 0; i < size; i++) {
        jstring j_string = (jstring)(*jenv)->GetObjectArrayElement(jenv, $input, i);
        const char * c_string = (*jenv)->GetStringUTFChars(jenv, j_string, 0);
        $1[i] = malloc(strlen((c_string)+1)*sizeof(const char *));
        strcpy($1[i], c_string);
        (*jenv)->ReleaseStringUTFChars(jenv, j_string, c_string);
        (*jenv)->DeleteLocalRef(jenv, j_string);
    }
    $1[i] = 0;
}

/* This cleans up the memory we malloc'd before the function call */
%typemap(freearg) char ** {
    jint i;
    for (i=0; i < size$jargnum-1; i++)
        free($1[i]);
    free($1);
}

/* This allows a C function to return a char ** as a Java String array */
%typemap(out) char ** {
    jint i;
    jint len=0;
    jstring temp_string;
    jclass clazz = (*jenv)->FindClass(jenv, "java/lang/String");

    while ($1[len]) len++;
    jresult = (*jenv)->NewObjectArray(jenv, len, clazz, NULL);
    /* exception checking omitted */

    for (i=0; i < len; i++) {
        temp_string = (*jenv)->NewStringUTF(jenv, *result++);
        (*jenv)->SetObjectArrayElement(jenv, jresult, i, temp_string);
        (*jenv)->DeleteLocalRef(jenv, temp_string);
    }
}

/* These 3 typemaps tell SWIG what JNI and Java types to use */
%typemap(jni) char ** "jobjectArray"
%typemap(jtype) char ** "String[]"
%typemap(jstype) char ** "String[]"

/* These 2 typemaps handle the conversion of the jtype to jstype typemap type
   and visa versa */
%typemap(javain) char ** "$javainput"
%typemap(javaout) char ** {
    return $jnicall;
}

/* Now a few test functions */
%inline %{
```

```

int print_args(char **argv) {
    int i = 0;
    while (argv[i]) {
        printf("argv[%d] = %s\n", i, argv[i]);
        i++;
    }
    return i;
}

char **get_args() {
    static char *values[] = { "Dave", "Mike", "Susan", "John", "Michelle", 0};
    return &values[0];
}

%}

```

Note that the 'C' JNI calling convention is used. Checking for any thrown exceptions after JNI function calls has been omitted. When this module is compiled, our wrapped C functions can be used by the following Java program:

```

// File main.java

public class main {

    static {
        try {
            System.loadLibrary("example");
        } catch (UnsatisfiedLinkError e) {
            System.err.println("Native code library failed to load. " + e);
            System.exit(1);
        }
    }

    public static void main(String argv[]) {
        String animals[] = {"Cat", "Dog", "Cow", "Goat"};
        example.print_args(animals);
        String args[] = example.get_args();
        for (int i=0; i<args.length; i++)
            System.out.println(i + ":" + args[i]);
    }
}

```

When compiled and run we get:

```

$ java main
argv[0] = Cat
argv[1] = Dog
argv[2] = Cow
argv[3] = Goat
0:Dave
1:Mike
2:Susan
3:John
4:Michelle

```

In the example, a few different typemaps are used. The "in" typemap is used to receive an input argument and convert it to a C array. Since dynamic memory allocation is used to allocate memory for the array, the "freearg" typemap is used to later release this memory after the execution of the C function. The "out" typemap is used for function return values. Lastly the "jni", "jtype" and "jstype" typemaps are also required to specify what Java types to use.

### 19.9.5 Expanding a Java object to multiple arguments

Suppose that you had a collection of C functions with arguments such as the following:

```
int foo(int argc, char **argv);
```

In the previous example, a typemap was written to pass a Java String array as the char `**argv`. This allows the function to be used from Java as follows:

```
example.foo(4, new String[]{"red", "green", "blue", "white"});
```

Although this works, it's a little awkward to specify the argument count. To fix this, a multi-argument typemap can be defined. This is not very difficult—you only have to make slight modifications to the previous example's typemaps:

```
%typemap(in) (int argc, char **argv) {
    int i = 0;
    $1 = (*jenv)->GetArrayLength(jenv, $input);
    $2 = (char **) malloc(($1+1)*sizeof(char *));
    /* make a copy of each string */
    for (i = 0; i < $1; i++) {
        jstring j_string = (jstring)(*jenv)->GetObjectArrayElement(jenv, $input, i);
        const char * c_string = (*jenv)->GetStringUTFChars(jenv, j_string, 0);
        $2[i] = malloc(strlen((c_string)+1)*sizeof(const char *));
        strcpy($2[i], c_string);
        (*jenv)->ReleaseStringUTFChars(jenv, j_string, c_string);
        (*jenv)->DeleteLocalRef(jenv, j_string);
    }
    $2[i] = 0;
}

%typemap(freearg) (int argc, char **argv) {
    int i;
    for (i=0; i < $1-1; i++)
        free($2[i]);
    free($2);
}

%typemap(jni) (int argc, char **argv) "jobjectArray"
%typemap(jtype) (int argc, char **argv) "String[]"
%typemap(jstype) (int argc, char **argv) "String[]"

%typemap(javain) (int argc, char **argv) "$javainput"
```

When writing a multiple-argument typemap, each of the types is referenced by a variable such as `$1` or `$2`. The typemap code simply fills in the appropriate values from the supplied Java parameter.

With the above typemap in place, you will find it no longer necessary to supply the argument count. This is automatically set by the typemap code. For example:

```
example.foo(new String[]{"red", "green", "blue", "white"});
```

## 19.9.6 Using typemaps to return arguments

A common problem in some C programs is that values may be returned in function parameters rather than in the return value of a function. The `typemaps.i` file defines `INPUT`, `OUTPUT` and `INOUT` typemaps which can be used to solve some instances of this problem. This library file uses an array as a means of moving data to and from Java when wrapping a C function that takes non const pointers or non const references as parameters.

Now we are going to outline an alternative approach to using arrays for C pointers. The `INOUT` typemap uses a `double[ ]` array for receiving and returning the `double*` parameters. In this approach we are able to use a Java class `myDouble` instead of `double[ ]` arrays where the C pointer `double*` is required.

Here is our example function:

```
/* Returns a status value and two values in out1 and out2 */
int spam(double a, double b, double *out1, double *out2);
```

If we define a structure `MyDouble` containing a `double` member variable and use some typemaps we can solve this problem. For example we could put the following through SWIG:

```
%module example

/* Define a new structure to use instead of double * */
%inline %{
typedef struct {
    double value;
} MyDouble;
%}

%{
/* Returns a status value and two values in out1 and out2 */
int spam(double a, double b, double *out1, double *out2) {
    int status = 1;
    *out1 = a*10.0;
    *out2 = b*100.0;
    return status;
};
%}

/*
This typemap will make any double * function parameters with name OUTVALUE take an
argument of MyDouble instead of double *. This will
allow the calling function to read the double * value after returning from the function.
*/
%typemap(in) double *OUTVALUE {
    jclass clazz = jenv->FindClass("MyDouble");
    jfieldID fid = jenv->GetFieldID(clazz, "swigCPtr", "J");
    jlong cPtr = jenv->GetLongField($input, fid);
    MyDouble *pMyDouble = NULL;
    *(MyDouble **)&pMyDouble = *(MyDouble **)&cPtr;
    $1 = &pMyDouble->value;
}

%typemap(jtype) double *OUTVALUE "MyDouble"
%typemap(jstype) double *OUTVALUE "MyDouble"
%typemap(jni) double *OUTVALUE "jobject"

%typemap(javain) double *OUTVALUE "$javainput"

/* Now we apply the typemap to the named variables */
%apply double *OUTVALUE { double *out1, double *out2 };
int spam(double a, double b, double *out1, double *out2);
```

Note that the C++ JNI calling convention has been used this time and so must be compiled as C++ and the `-c++` commandline must be passed to SWIG. JNI error checking has been omitted for clarity.

What the typemaps do are make the named `double*` function parameters use our new `MyDouble` wrapper structure. The "in" typemap takes this structure, gets the C++ pointer to it, takes the `double` value member variable and passes it to the C++ `spam` function. In Java, when the function returns, we use the SWIG created `getValue()` function to get the output value. The following Java program demonstrates this:

```
// File: main.java

public class main {

    static {
        try {
            System.loadLibrary("example");
        } catch (UnsatisfiedLinkError e) {
            System.err.println("Native code library failed to load. " + e);
            System.exit(1);
        }
    }
}
```



```

    }
}

public static void main(String argv[]) {
    MyDouble out1 = new MyDouble();
    MyDouble out2 = new MyDouble();
    int ret = example.spam(1.2, 3.4, out1, out2);
    System.out.println(ret + " " + out1.getValue() + " " + out2.getValue());
}
}

```

When compiled and run we get:

```

$ java main
1 12.0 340.0

```

### 19.9.7 Adding Java downcasts to polymorphic return types

SWIG support for polymorphism works in that the appropriate virtual function is called. However, the default generated code does not allow for downcasting. Let's examine this with the follow code:

```

#include "std_string.i"

#include <iostream>
using namespace std;
class Vehicle {
public:
    virtual void start() = 0;
    ...
};

class Ambulance : public Vehicle {
    string vol;
public:
    Ambulance(string volume) : vol(volume) {}
    virtual void start() {
        cout << "Ambulance started" << endl;
    }
    void sound_siren() {
        cout << vol << " siren sounded!" << endl;
    }
    ...
};

Vehicle *vehicle_factory() {
    return new Ambulance("Very loud");
}

```

If we execute the following Java code:

```

Vehicle vehicle = example.vehicle_factory();
vehicle.start();

Ambulance ambulance = (Ambulance)vehicle;
ambulance.sound_siren();

```

We get:

```

Ambulance started
java.lang.ClassCastException
    at main.main(main.java:16)

```

Even though we know from examination of the C++ code that `vehicle_factory` returns an object of type `Ambulance`, we are not able to use this knowledge to perform the downcast in Java. This occurs because the runtime type information is not

completely passed from C++ to Java when returning the type from `vehicle_factory()`. Usually this is not a problem as virtual functions do work by default, such as in the case of `start()`. There are a few solutions to getting downcasts to work.

The first is not to use a Java cast but a call to C++ to make the cast. Add this to your code:

```
%exception Ambulance::dynamic_cast(Vehicle *vehicle) {
    $action
    if (!result) {
        jclass excep = jenv->FindClass("java/lang/ClassCastException");
        if (excep) {
            jenv->ThrowNew(excep, "dynamic_cast exception");
        }
    }
}
%extend Ambulance {
    static Ambulance *dynamic_cast(Vehicle *vehicle) {
        return dynamic_cast<Ambulance *>(vehicle);
    }
};
```

It would then be used from Java like this

```
Ambulance ambulance = Ambulance.dynamic_cast(vehicle);
ambulance.sound_siren();
```

Should `vehicle` not be of type `ambulance` then a Java `ClassCastException` is thrown. The next solution is a purer solution in that Java downcasts can be performed on the types. Add the following before the definition of `vehicle_factory`:

```
%typemap(out) Vehicle * {
    Ambulance *downcast = dynamic_cast<Ambulance *>($1);
    *(Ambulance **)&$result = downcast;
}

%typemap(javaout) Vehicle * {
    return new Ambulance($jnicall, $owner);
}
```

Here we are using our knowledge that `vehicle_factory` always returns type `Ambulance` so that the Java proxy is created as a type `Ambulance`. If `vehicle_factory` can manufacture any type of `Vehicle` and we want to be able to downcast using Java casts for any of these types, then a different approach is needed. Consider expanding our example with a new `Vehicle` type and a more flexible factory function:

```
class FireEngine : public Vehicle {
public:
    FireEngine() {}
    virtual void start() {
        cout << "FireEngine started" << endl;
    }
    void roll_out_hose() {
        cout << "Hose rolled out" << endl;
    }
    ...
};

Vehicle *vehicle_factory(int vehicle_number) {
    if (vehicle_number == 0)
        return new Ambulance("Very loud");
    else
        return new FireEngine();
}
```

To be able to downcast with this sort of Java code:

```
FireEngine fireengine = (FireEngine)example.vehicle_factory(1);
```

```

fireengine.roll_out_hose();
Ambulance ambulance = (Ambulance)example.vehicle_factory(0);
ambulance.sound_siren();

```

the following typemaps targeted at the `vehicle_factory` function will achieve this. Note that in this case, the Java class is constructed using JNI code rather than passing a pointer across the JNI boundary in a Java long for construction in Java code.

```

%typemap(jni) Vehicle *vehicle_factory "jobject"
%typemap(jtype) Vehicle *vehicle_factory "Vehicle"
%typemap(jstype) Vehicle *vehicle_factory "Vehicle"
%typemap(javaout) Vehicle *vehicle_factory {
    return $jnicall;
}

%typemap(out) Vehicle *vehicle_factory {
    Ambulance *ambulance = dynamic_cast<Ambulance *>($1);
    FireEngine *fireengine = dynamic_cast<FireEngine *>($1);
    if (ambulance) {
        // call the Ambulance(long cPtr, boolean cMemoryOwn) constructor
        jclass clazz = jenv->FindClass("Ambulance");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(Ambulance **)&cptr = ambulance;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    }
    else if (fireengine) {
        // call the FireEngine(long cPtr, boolean cMemoryOwn) constructor
        jclass clazz = jenv->FindClass("FireEngine");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(FireEngine **)&cptr = fireengine;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    }
    else {
        cout << "Unexpected type " << endl;
    }
    if (!$result)
        cout << "Failed to create new java object" << endl;
}

```

Better error handling would need to be added into this code. There are other solutions to this problem, but this last example demonstrates some more involved JNI code. SWIG usually generates code which constructs the proxy classes using Java code as it is easier to handle error conditions and is faster. Note that the JNI code above uses a number of string lookups to call a constructor, whereas this would not occur using byte compiled Java code.

### 19.9.8 Adding an equals method to the Java classes

When a pointer is returned from a JNI function, it is wrapped using a new Java proxy class or type wrapper class. Even when the pointers are the same, it will not be possible to know that the two Java classes containing those pointers are actually the same object. It is common in Java to use the `equals()` method to check whether two objects are equivalent. An equals method is easily added to all proxy classes. For example:

```

%typemap(javacode) SWIGTYPE %{
    public boolean equals(Object obj) {
        boolean equal = false;
        if (obj instanceof $javaclassname)

```

```

        equal = ((($javaclassname)obj).swigCPtr == this.swigCPtr);
        return equal;
    }
}%

class Foo { };
Foo* returnFoo(Foo *foo) { return foo; }

```

The following would display false without the `javacode` typemap above. With the typemap defining the `equals` method the result is true.

```

Foo foo1 = new Foo();
Foo foo2 = example.returnFoo(foo1);
System.out.println("foo1? " + foo1.equals(foo2));

```

### 19.9.9 Void pointers and a common Java base class

One might wonder why the common code that SWIG emits for the proxy and type wrapper classes is not pushed into a base class. The reason is that although `swigCPtr` could be put into a common base class for all classes wrapping C structures, it would not work for C++ classes involved in an inheritance chain. Each class derived from a base needs a separate `swigCPtr` because C++ compilers sometimes use a different pointer value when casting a derived class to a base. Additionally as Java only supports single inheritance, it would not be possible to derive wrapped classes from your own pure Java classes if the base class has been 'used up' by SWIG. However, you may want to move some of the common code into a base class. Here is an example which uses a common base class for all proxy classes and type wrapper classes:

```

%typemap(javabase) SWIGTYPE, SWIGTYPE *, SWIGTYPE &, SWIGTYPE [],
                    SWIGTYPE (CLASS::*) "SWIG"

%typemap(javacode) SWIGTYPE, SWIGTYPE *, SWIGTYPE &, SWIGTYPE [],
                    SWIGTYPE (CLASS::*) %{
    protected long getPointer() {
        return swigCPtr;
    }
}%

```

Define new base class called SWIG:

```

public abstract class SWIG {
    protected abstract long getPointer();

    public boolean equals(Object obj) {
        boolean equal = false;
        if (obj instanceof SWIG)
            equal = (((SWIG)obj).getPointer() == this.getPointer());
        return equal;
    }

    SWIGTYPE_p_void getVoidPointer() {
        return new SWIGTYPE_p_void(getPointer(), false);
    }
}

```

This example contains some useful functionality which you may want in your code.

- It has an `equals()` method. Unlike the previous example, the method code isn't replicated in all classes.
- It also has a function which effectively implements a cast from the type of the proxy/type wrapper class to a void pointer. This is necessary for passing a proxy class or a type wrapper class to a function that takes a void pointer.

## 19.10 Living with Java Directors

This section is intended to address frequently asked questions and frequently encountered problems when using Java directors.

1. *When my program starts up, it complains that method\_foo cannot be found in a Java method called swig\_module\_init. How do I fix this?*

Open up the C++ wrapper source code file and look for "method\_foo" (include the double quotes, they are important!) Look at the JNI field descriptor and make sure that each class that occurs in the descriptor has the correct package name in front of it. If the package name is incorrect, put a "javapackage" typemap in your SWIG interface file.

2. *I'm compiling my code and I'm using templates. I provided a javapackage typemap, but SWIG doesn't generate the right JNI field descriptor.*

Use the template's renamed name as the argument to the "javapackage" typemap:

```
%typemap(javapackage) std::vector<int> "your.package.here"
%template(VectorOfInt) std::vector<int>;
```

3. *When I pass class pointers or references through a C++ upcall and I try to type cast them, Java complains with a ClassCastException. What am I doing wrong?*

Normally, a non-director generated Java proxy class creates temporary Java objects as follows:

```
public static void MyClass_method_upcall(MyClass self, long jarg1)
{
    Foo darg1 = new Foo(jarg1, false);

    self.method_upcall(darg1);
}
```

Unfortunately, this loses the Java type information that is part of the underlying Foo director proxy class's java object pointer causing the type cast to fail. The SWIG Java module's director code attempts to correct the problem, **but only for director-enabled classes**, since the director class retains a global reference to its Java object. Thus, for director-enabled classes **and only for director-enabled classes**, the generated proxy Java code looks something like:

```
public static void MyClass_method_upcall(MyClass self, long jarg1,
                                         Foo jarg1_object)
{
    Foo darg1 = (jarg1_object != null ? jarg1_object : new Foo(jarg1, false));

    self.method_upcall(darg1);
}
```

When you import a SWIG interface file containing class definitions, the classes you want to be director-enabled must be have the `feature("director")` enabled for type symmetry to work. This applies even when the class being wrapped isn't a director-enabled class but takes parameters that are director-enabled classes.

The current "type symmetry" design will work for simple C++ inheritance, but will most likely fail for anything more complicated such as tree or diamond C++ inheritance hierarchies. Those who are interested in challenging problems are more than welcome to hack the `Java::Java_director_declaration` method in `Source/Modules/java.cxx`.

If all else fails, you can use the `downcastXXXXX()` method to attempt to recover the director class's Java object pointer. For the Java Foo proxy class, the Foo director class's java object pointer can be accessed through the `javaObjectFoo()` method. The generated method's signature is:

```
public static Foo javaObjectFoo(Foo obj);
```

From your code, this method is invoked as follows:

```
public class MyClassDerived {
    public void method_upcall(Foo foo_object)
    {
        FooDerived    derived = (foo_object != null ?
                                (FooDerived) Foo.downcastFoo(foo_object) : null);
        /* rest of your code here */
    }
}
```

An good approach for managing downcasting is placing a static method in each derived class that performs the downcast from the superclass, e.g.,

```
public class FooDerived extends Foo {
    /* ... */
    public static FooDerived downcastFooDerived(Foo foo_object)
    {
        try {
            return (foo_object != null ? (FooDerived) Foo.downcastFoo(foo_object);
        }

        catch (ClassCastException exc) {
            // Wasn't a FooDerived object, some other subclass of Foo
            return null;
        }
    }
}
```

Then change the code in `MyClassDerived` as follows:

```
public class MyClassDerived extends MyClass {
    /* ... */
    public void method_upcall(Foo foo_object)
    {
        FooDerived    derived = FooDerived.downcastFooDerived(foo_object) : null);
        /* rest of your code here */
    }
}
```

#### 4. *Why isn't the proxy class declared abstract? Why aren't the director upcall methods in the proxy class declared abstract?*

Declaring the proxy class and its methods abstract would break the JNI argument marshalling and SWIG's downcall functionality (going from Java to C++.) Create an abstract Java subclass that inherits from the director-enabled class instead. Using the previous `Foo` class example:

```
public abstract class UserVisibleFoo extends Foo {
    /** Make sure user overrides this method, it's where the upcall
     * happens.
     */
    public abstract void method_upcall(Foo foo_object);

    /// Downcast from Foo to UserVisibleFoo
    public static UserVisibleFoo downcastUserVisibleFoo(Foo foo_object)
    {
        try {
            return (foo_object != null ? (FooDerived) Foo.downcastFoo(foo_object) : null);
        }

        catch (ClassCastException exc) {
            // Wasn't a FooDerived object, some other subclass of Foo
            return null;
        }
    }
}
```

This doesn't prevent the user from creating subclasses derived from Foo, however, UserVisibleFoo provides the safety net that reminds the user to override the `method_upcall()` method.

## 19.11 Odds and ends

### 19.11.1 JavaDoc comments

The SWIG documentation system is currently deprecated. When it is resurrected JavaDoc comments will be fully supported. If you can't wait for the full documentation system a couple of workarounds are available. The `%javamethodmodifiers` feature can be used for adding proxy class method comments and module class method comments. The "javainports" typemap can be hijacked for adding in proxy class JavaDoc comments. The `jniclassimports` or `jniclassclassmodifiers` pragmas can also be used for adding intermediary JNI class comments and likewise the `moduleimports` or `moduleclassmodifiers` pragmas for the module class. Here is an example adding in a proxy class and method comment:

```
%javamethodmodifiers Barmy::lose_marbles() "
/**
 * Calling this method will make you mad.
 * Use with <b>utmost</b> caution.
 */
public";

%typemap(javainports) Barmy "
/** The crazy class. Use as a last resort. */"

class Barmy {
public:
    void lose_marbles() {}
};
```

Note the "public" added at the end of the `%javamethodmodifiers` as this is the default for this feature. The generated proxy class with JavaDoc comments is then as follows:

```
/** The crazy class. Use as a last resort. */
public class Barmy {
...
/**
 * Calling this method will make you mad.
 * Use with <b>utmost</b> caution.
 */
    public void lose_marbles() {
        ...
    }
...
}
```

### 19.11.2 Functional interface without proxy classes

It is possible to run SWIG in a mode that does not produce proxy classes by using the `-noproxy` commandline option. The interface is rather primitive when wrapping structures or classes and is accessed through function calls to the module class. All the functions in the module class are wrapped by functions with identical names as those in the intermediary JNI class.

Consider the example we looked at when examining proxy classes:

```
class Foo {
public:
    int x;
    int spam(int num, Foo* foo);
};
```

When using `-noproxy`, type wrapper classes are generated instead of proxy classes. Access to all the functions and variables is

through a C like set of functions where the first parameter passed is the pointer to the class, that is an instance of a type wrapper class. Here is what the module class looks like:

```
public class example {
    public static void set_Foo_x(SWIGTYPE_p_Foo self, int x) {...}
    public static int get_Foo_x(SWIGTYPE_p_Foo self) {...}
    public static int Foo_spam(SWIGTYPE_p_Foo self, int num, SWIGTYPE_p_Foo foo) {...}
    public static SWIGTYPE_p_Foo new_Foo() {...}
    public static void delete_Foo(SWIGTYPE_p_Foo self) {...}
}
```

This approach is not nearly as natural as using proxy classes as the functions need to be used like this:

```
SWIGTYPE_p_Foo foo = example.new_Foo();
example.set_Foo_x(foo, 10);
int var = example.get_Foo_x(foo);
example.Foo_spam(foo, 20, foo);
example.delete_Foo(foo);
```

Unlike proxy classes, there is no attempt at tracking memory. All destructors have to be called manually for example the `delete_Foo(foo)` call above.

### 19.11.3 Using your own JNI functions

You may have some hand written JNI functions that you want to use in addition to the SWIG generated JNI functions. Adding these to your SWIG generated package is possible using the `%native` directive. If you don't want SWIG to wrap your JNI function then of course you can simply use the `%ignore` directive. However, if you want SWIG to generate just the Java code for a JNI function then use the `%native` directive. The C types for the parameters and return type must be specified in place of the JNI types and the function name must be the native method name. For example:

```
%native (HandRolled) void HandRolled(int, char *);
%{
JNIEXPORT void JNICALL Java_packageName_moduleName_HandRolled(JNIEnv *, jclass,
                                                                    jlong, jstring);
%}
```

No C JNI function will be generated and the `Java_packageName_moduleName_HandRolled` function will be accessible using the SWIG generated Java native method call in the intermediary JNI class which will look like this:

```
public final static native void HandRolled(int jarg1, String jarg2);
```

and as usual this function is wrapped by another which for a global C function would appear in the module class:

```
public static void HandRolled(int arg0, String arg1) {
    exampleJNI.HandRolled(arg0, arg1);
}
```

The `packageName` and `moduleName` must of course be correct else you will get linker errors when the JVM dynamically loads the JNI function. You may have to add in some "jtype", "jstype", "javain" and "javaout" typemaps when wrapping some JNI types. Here the default typemaps work for `int` and `char *`.

In summary the `%native` directive is telling SWIG to generate the Java code to access the JNI C code, but not the JNI C function itself. This directive is only really useful if you want to mix your own hand crafted JNI code and the SWIG generated code into one Java class or package.

### 19.11.4 Performance concerns and hints

If you're directly manipulating huge arrays of complex objects from Java, performance may suffer greatly when using the array functions in `arrays_java.i`. Try and minimise the expensive JNI calls to C/C++ functions, perhaps by using temporary Java



variables instead of accessing the information directly from the C/C++ object.

Java classes without any finalizers generally speed up code execution as there is less for the garbage collector to do. Finalizer generation can be stopped by using an empty `javafinalize` typemap:

```
%typemap(javafinalize) SWIGTYPE " "
```

However, you will have to be careful about memory management and make sure that you code in a call to the `delete()` member function. This method calls the C++ destructor or `free()` for C code.

## 19.12 Examples

The directory `Examples/java` has a number of further examples. Take a look at these if you want to see some of the techniques described in action. The `Examples/index.html` file in the parent directory contains the SWIG Examples Documentation and is a useful starting point. If your SWIG installation went well Unix users should be able to type `make` in each example directory, then `java main` to see them running. For the benefit of Windows users, there are also Visual C++ project files in a couple of the [Windows Examples](#).

## 20 SWIG and Modula-3

- [Overview](#)
  - ◆ [Why not scripting ?](#)
  - ◆ [Why Modula-3 ?](#)
  - ◆ [Why C / C++ ?](#)
  - ◆ [Why SWIG ?](#)
- [Conception](#)
  - ◆ [Interfaces to C libraries](#)
  - ◆ [Interfaces to C++ libraries](#)
- [Preliminaries](#)
  - ◆ [Compilers](#)
  - ◆ [Additional Commandline Options](#)
- [Modula-3 typemaps](#)
  - ◆ [Inputs and outputs](#)
  - ◆ [Subranges, Enumerations, Sets](#)
  - ◆ [Objects](#)
  - ◆ [Imports](#)
  - ◆ [Exceptions](#)
  - ◆ [Example](#)
- [More hints to the generator](#)
  - ◆ [Features](#)
  - ◆ [Pragmas](#)
- [Remarks](#)

This chapter describes SWIG's support of [Modula-3](#). You should be familiar with the [basics](#) of SWIG, especially [typemaps](#).

### 20.1 Overview

The Modula-3 support is very basic and highly experimental! Many features are still not designed satisfyingly and I need more discussion about the odds and ends. Don't rely on any feature, incompatible changes are likely in the future! The Modula-3 generator was already useful for interfacing to the libraries

1. [PLPlot](#)
2. [FTW](#) .

I took some more time to explain why I think it's right what I'm doing. So the introduction got a bit longer than it should ... ;—)

#### 20.1.1 Why not scripting ?

SWIG started as wrapper from the fast compiled languages C and C++ to high level scripting languages like Python. Although scripting languages are designed to make programming life easier by hiding machine internals from the programmer there are several aspects of todays scripting languages that are unfavourable in my opinion.

Besides C, C++, Cluster (a Modula derivate for Amiga computers) I evaluated several scripting like languages in the past: Different dialects of BASIC, Perl, ARexx (a variant of Rexx for Amiga computers), shell scripts. I found them too inconsistent, too weak in distinguishing types, too weak in encapsulating pieces of code. Eventually I have started several projects in Python because of the fine syntax. But when projects became larger I lost the track. I got convinced that one can not have maintainable code in a language that is not statically typed. In fact the main advantages of scripting languages e.g. matching regular expressions, complex built-in datatypes like lists, dictionaries, are not advantages of the language itself but can be provided by function libraries.

## 20.1.2 Why Modula-3 ?

Modula-3 is a compiler language in the tradition of Niklaus Wirth's Modula 2, which is in turn a successor of the popular Pascal. I have chosen Modula-3 because of its logical syntax, strong modularization, the type system which is very detailed for machine types compared to other languages. Of course it supports all of the modern games like exceptions, objects, garbage collection, threads. While C++ programmers must control three languages, namely the preprocessor, C and ++, Modula-3 is made in one go and the language definition is really compact.

On the one hand Modula-3 can be safe (but probably less efficient) in normal modules while providing much static and dynamic safety. On the other hand you can write efficient but less safe code in the style of C within UNSAFE modules.

Unfortunately Modula's safety and strength requires more writing than scripting languages do. Today if I want to save characters I prefer Haskell (similar to OCAML) – it's statically typed, too.

## 20.1.3 Why C / C++ ?

Although it is no problem to write Modula-3 programs that performs as fast as C most libraries are not written in Modula-3 but in C. Fortunately the binary interface of most function libraries can be addressed by Modula-3. Even more fortunately even non-C libraries may provide C header files. This is where SWIG becomes helpful.

## 20.1.4 Why SWIG ?

The C headers and the possibility to interface to C libraries still leaves the work for you to write Modula-3 interfaces to them. To make things comfortable you will also need wrappers that convert between high-level features of Modula-3 (garbage collecting, exceptions) and the low level of the C libraries.

SWIG converts C headers to Modula-3 interfaces for you. You could call the C functions without loss of efficiency but it won't be joy because you could not pass TEXTs or open arrays and you would have to process error return codes rather than exceptions. But using some typemaps SWIG will also generate wrappers that bring the whole Modula-3 comfort to you. If the library API is ill designed writing appropriate typemaps can be still time-consuming. E.g. C programmers are very creative to work-around missing data types like (real) enumerations and sets. You should turn such work-arounds back to the Modula-3 way otherwise you lose static safety and consistency.

But you have still a problem: C library interfaces are often ill. They lack for certain information because C compilers wouldn't care about. You should integrate detailed type information by adding typedefs and consts and you should persuade the C library programmer to add this information to his interface. Only this way other language users can benefit from your work and only this way you can easily update your interfaces when a new library version is released. You will realise that writing **good** SWIG interfaces is very costly and it will only amortise when considering evolving libraries.

Without SWIG you would probably never consider to call C++ libraries from Modula-3. But with SWIG this is worth a consideration. SWIG can write C wrappers to C++ functions and object methods that may throw exceptions. In fact it breaks down C++ libraries to C interfaces which can be in turn called from Modula-3. To make it complete you can hide the C interface with Modula-3 classes and exceptions.

Although SWIG does the best it can do it can only serve as a one-way strategy. That means you can use C++ libraries with Modula-3 (even with call back functions), but it's certainly not possible to smoothly integrate Modula-3 code into a C / C++ project.

## 20.2 Conception

### 20.2.1 Interfaces to C libraries

Modula-3 has an integrated support for calling C functions. This is also extensively used by the standard Modula-3 libraries to call OS functions. The Modula-3 part of SWIG and the corresponding SWIG library [modula3.swg](#) contain code that uses these features. Because of the built-in support there is no need for calling the SWIG kernel to generate wrappers written in C. All

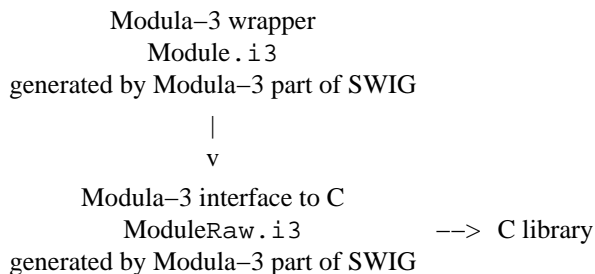
conversion and argument checking can be done in Modula-3 and the interfacing is quite efficient. All you have to do is to write pieces of Modula-3 code that SWIG puts together.

C library support integrated in Modula-3	
Pragma <* EXTERNAL *>	Precedes a declaration of a PROCEDURE that is implemented in an external library instead of a Modula-3 module.
Pragma <* CALLBACK *>	Precedes a declaration of a PROCEDURE that should be called by external library code.
Module Ctypes	Contains Modula-3 types that match some basic C types.
Module M3toC	Contains routines that convert between Modula-3's TEXT type and C's char * type.

In each run of SWIG the Modula-3 part generates several files:

Module name scheme	Identifier for %insert	Description
ModuleRaw.i3	m3rawintf	Declaration of types that are equivalent to those of the C library, EXTERNAL procedures as interface to the C library functions
ModuleRaw.m3	m3rawimpl	Almost empty.
Module.i3	m3wrapintf	Declaration of comfortable wrappers to the C library functions.
Module.m3	m3wrapimpl	Implementation of the wrappers that convert between Modula-3 and C types, check for validity of values, hand-over resource management to the garbage collector using WeakRefs and raises exceptions.
m3makefile	m3makefile	Add the modules above to the Modula-3 project and specify the name of the Modula-3 wrapper library to be generated. Today I'm not sure if it is a good idea to create a m3makefile in each run, because SWIG must be started for each Modula-3 module it creates. Thus the m3makefile is overwritten each time. :-(

Here's a scheme of how the function calls to Modula-3 wrappers are redirected to C library functions:



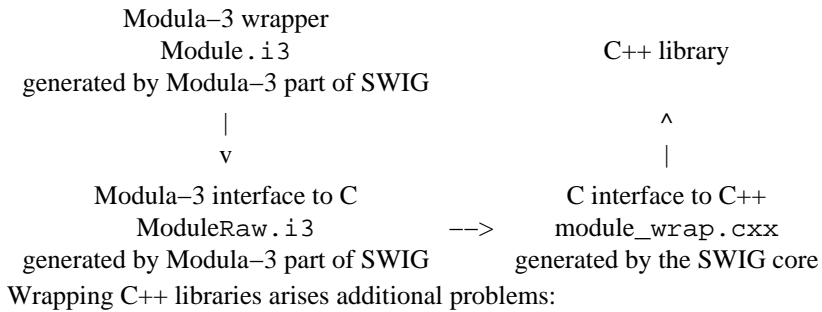
I have still no good conception how one can split C library interfaces into type oriented interfaces. A Module in Modula-3 represents an Abstract DataType (or call it a static classes, i.e. a class without virtual methods). E.g. if you have a principal type, say Database, it is good Modula-3 style to set up one Module with the name Database where the database type is declared with the name T and where all functions are declared that operates on it.

The normal operation of SWIG is to generate a fixed set of files per call. To generate multiple modules one has to write one SWIG interface (different SWIG interfaces can share common data) per module. Identifiers belonging to a different module may ignored (%ignore) and the principal type must be renamed (%typemap).

## 20.2.2 Interfaces to C++ libraries

Interfaces to C++ files are much more complicated and there are some more design decisions that are not made, yet. Modula-3 has no support for C++ functions but C++ compilers should support generating C++ functions with a C interface.

Here's a scheme of how the function calls to Modula-3 wrappers a redirected to C library functions:



- Is it sensible to wrap C++ classes with Modula-3 classes?
- How to find the wrapping Modula-3 class for a class pointer that is returned by a C++ routine?
- How to deal with multiple inheritance which was neglected for Modula-3 for good reasons?
- Is it possible to sub-class C++ classes with Modula-3 code? This issue is addressed by directors, a feature that was experimentally added to some Language modules like [Java](#) and [Python](#).
- How to manage storage with the garbage collector of Modula-3? Support for [%newobject and %typemap\(newfree\)](#) isn't implemented, yet. What's about resources that are managed by the garbage collector but shall be passed back to the storage management of the C++ library? This is a general issue which is not solved in a satisfying fashion as far as I know.
- How to turn C++ exceptions into Modula-3 exceptions? There's also no support for [%exception](#), yet.

Be warned: There is no C++ library I wrote a SWIG interface for, so I'm not sure if this is possible or sensible, yet.

## 20.3 Preliminaries

### 20.3.1 Compilers

There are different Modula-3 compilers around: `cm3`, `pm3`, `ezm3`, Klagenfurth Modula-3, Cambridge Modula-3. SWIG itself does not contain compiler specific code but the library file [modula3.swg](#) may do so. For testing examples I use Critical Mass `cm3`.

### 20.3.2 Additional Commandline Options

There are some experimental command line options that prevent SWIG from generating interface files. Instead files are emitted that may assist you when writing SWIG interface files.

Modula-3 specific options	Description
<code>-generateconst &lt;file&gt;</code>	Disable generation of interfaces and wrappers. Instead write code for computing numeric values of constants to the specified file. C code may contain several constant definitions written as preprocessor macros. Other language modules of SWIG use compute-once-use-readonly variables or functions to wrap such definitions. All of them can invoke C code dynamically for computing the macro values. But if one wants to turn them into Modula-3 integer constants, enumerations or set types, the values of these expressions has to be known statically. Although definitions like <code>(1 &lt;&lt; FLAG_MAXIMIZEWINDOW)</code> must be considered as good C style they are hard to convert to Modula-3 since the value computation can use every feature of C. Thus I implemented these switch to extract all constant definitions and write a C program that output the values of them. It works for numeric constants only and treats all of them as <code>double</code> . Future versions may generate a C++ program that can detect the type of the macros by overloaded output functions. Then strings can also be processed.
<code>-generaterename &lt;file&gt;</code>	Disable generation of interfaces and wrappers. Instead generate suggestions for <code>%rename</code> . C libraries use a naming style that is neither homogenous nor similar to that of Modula-3. C function names often contain a prefix denoting the library and some name components separated by underscores or capitalization changes. To get library interfaces that are really Modula-3 like you should rename the

	function names with the <code>%rename</code> directive. This switch outputs a list of such directives with a name suggestion generated by a simple heuristic.
<code>-generatetypemap</code> <code>&lt;file&gt;</code>	Disable generation of interfaces and wrappers. Instead generate templates for some basic typemaps.

## 20.4 Modula-3 typemaps

### 20.4.1 Inputs and outputs

Each C procedure has a bunch of inputs and outputs. Inputs are passed as function arguments, outputs are updated referential arguments and the function value.

Each C type can have several typemaps that apply only in case if a type is used for an input argument, for an output argument, or for a return value. A further typemap may specify the direction that is used for certain parameters. I have chosen this separation in order to be able to write general typemaps for the typemap library [modula3.swg](#). In the library code the final usage of the type is not known. Using separate typemaps for each possible use allows appropriate definitions for each case. If these pre-definitions are fine then the direction of the function parameter is the only hint the user must give.

The typemaps specific to Modula-3 have a common name scheme: A typemap name starts with "m3", followed by "raw" or "wrap" depending on whether it controls the generation of the `ModuleRaw.i3` or the `Module.i3`, respectively. It follows an "in" for typemaps applied to input argument, "out" for output arguments, "arg" for all kind of arguments, "ret" for returned values.

The main task of SWIG is to build wrapper function, i.e. functions that convert values between C and Modula-3 and call the corresponding C function. Modula-3 wrapper functions generated by SWIG consist of the following parts:

- Generate PROCEDURE signature.
- Declare local variables.
- Convert input values from Modula-3 to C.
- Check for input value integrity.
- Call the C function.
- Check returned values, e.g. error codes.
- Convert and write back values into Modula-3 records.
- Free temporary storage.
- Return values.

Typemap	Example	Description
m3wrapargvar	<code>\$l: INTEGER := \$l_name;</code>	Declaration of some variables needed for temporary results.
m3wrapargconst	<code>\$l = "\$l_name";</code>	Declaration of some constant, maybe for debug purposes.
m3wrapargraw	<code>ORD(\$l_name)</code>	The expression that should be passed as argument to the raw Modula-3 interface function.
m3wrapargdir	<code>out</code>	Referential arguments can be used for input, output, update. ???
m3wrapinmode	<code>READONLY</code>	One of Modula-3 parameter modes VALUE (or empty), VAR, READONLY
m3wrapinname		New name of the input argument.
m3wrapintype		Modula-3 type of the input argument.
m3wrapindefault		Default value of the input argument
m3wrapinconv	<code>\$l := M3toC.SharedTtoS(\$l_name);</code>	Statement for converting the Modula-3 input value to C compliant value.
m3wrapincheck	<code>IF Text.Length(\$l_name) &gt; 10 THEN RAISE E("str too long"); END;</code>	Check the integrity of the input value.

m3wrapoutname		Name of the RECORD field to be used for returning multiple values. This applies to referential output arguments that shall be turned into return values.
m3wrapouttype		Type of the value that is returned instead of a referential output argument.
m3wrapoutconv		
m3wrapoutcheck		
m3wrapretraw		
m3wrapretrname		
m3wrapretrtype		
m3wrapretrvar		
m3wrapretrconv		
m3wrapretrcheck		
m3wrapfreearg	M3toC.FreeSharedS(str, arg1);	Free resources that were temporarily used in the wrapper. Since this step should never be skipped, SWIG will put it in the FINALLY branch of a TRY . . FINALLY structure.

## 20.4.2 Subranges, Enumerations, Sets

Subranges, enumerations, and sets are machine oriented types that make Modula very strong and expressive compared with the type systems of many other languages.

- Subranges are used for statically restricted choices of integers.
- Enumerations are used for named choices.
- Sets are commonly used for flag (option) sets.

Using them extensively makes Modula code very safe and readable. C supports enumerations, too, but they are not as safe as the ones of Modula. Thus they are abused for many things: For named choices, for integer constant definitions, for sets. To make it complete every way of defining a value in C (#define, const int, enum) is somewhere used for defining something that must be handled completely different in Modula-3 (INTEGER, enumeration, SET). I played around with several %features and %pragmas that split the task up into converting the C bit patterns (integer or bit set) into Modula-3 bit patterns (integer or bit set) and change the type as requested. See the corresponding [example](#). This is quite messy and not satisfying. So the best what you can currently do is to rewrite constant definitions manually. Though this is a tedious work that I'd like to automate.

## 20.4.3 Objects

Declarations of C++ classes are mapped to OBJECT types while it is tried to retain the access hierarchy "public – protected – private" using partial revelation. Though the [implementation](#) is not really useful, yet.

## 20.4.4 Imports

Pieces of Modula-3 code provided by typemaps may contain identifiers from foreign modules. If the typemap m3wrapinconv for blah \* contains code using the function M3toC.SharedTtoS you may declare %typemap("m3wrapinconv:import") blah \* %{M3toC%}. Then the module M3toC is imported if the m3wrapinconv typemap for blah \* is used at least once. Use %typemap("m3wrapinconv:import") blah \* %{MyConversions AS M3toC%} if you need module renaming. Unqualified import is not supported.

It is cumbersome to add this typemap to each piece of Modula-3 code. It is especially useful when writing general typemaps for the typemap library [modula3.swg](#). For a monolithic module you might be better off if you add the imports directly:

```
%insert(m3rawintf) %{
IMPORT M3toC;
%}
```

## 20.4.5 Exceptions

Modula-3 provides another possibility of an output of a function: exceptions. Any piece of Modula-3 code that SWIG inserts due to a typemap can raise an exception. This way you can also convert an error code from a C function into a Modula-3 exception. The RAISES clause is controlled by typemaps with the throws extension. If the typemap `m3wrapinconv` for `blah` \* contains code that may raise the exceptions `OSError.E` you should declare `%typemap( "m3wrapinconv:throws" )` `blah * % {OSError.E%}`.

## 20.4.6 Example

The generation of wrappers in Modula-3 needs very fine control to take advantage of the language features. Here is an example of a generated wrapper where almost everything is generated by a typemap:

```

      (* %relabel  m3wrapinmode m3wrapinname m3wrapintype  m3wrapindefault *)
PROCEDURE Name      (READONLY      str      :      TEXT      :=      ""      )
      (* m3wrapoutcheck:throws *)
: NameResult RAISES {E} =
CONST
  arglname = "str";                                (* m3wrapargconst *)
VAR
  arg0      : C.char_star;                          (* m3wrapretvar *)
  arg1      : C.char_star;                          (* m3wrapargvar *)
  arg2      : C.int;
  result    : RECORD
    (*m3wrapretname  m3wraprettype*)
    unixPath : TEXT;
    (*m3wrapoutname  m3wrapouttype*)
    checksum  : CARDINAL;
  END;
BEGIN
  TRY
    arg1 := M3toC.SharedTtoS(str);  (* m3wrapinconv *)
    IF Text.Length(arg1) > 10 THEN  (* m3wrapincheck *)
      RAISE E("str too long");
    END;
    (* m3wrappretraw      m3wrapargraw *)
    arg0 := MessyToUnix (arg1,  arg2);
    result.unixPath := M3toC.CopyStoT(arg0);  (* m3wrapretconv *)
    result.checksum := arg2;                  (* m3wrapoutconv *)
    IF result.checksum = 0 THEN               (* m3wrapoutcheck *)
      RAISE E("invalid checksum");
    END;
  FINALLY
    M3toC.FreeSharedS(str,arg1);  (* m3wrapfreearg *)
  END;
END Name;

```

## 20.5 More hints to the generator

### 20.5.1 Features

Feature	Example	Description
multiretval	<pre>%m3multiretval get_box; or %feature( "modula3:multiretval" ) get_box;</pre>	Let the denoted function return a RECORD rather than a plain value. This RECORD contains all arguments with "out" direction including the return value of the C function (if there is one). If more than one argument is "out" then the function <b>must</b> have the <code>multiretval</code> feature activated, but it is explicitly requested from the user to prevent



		mistakes.
constnumeric	<code>%constnumeric(12) twelve; or %feature("constnumeric","12") twelve;</code>	This feature can be used to tell Modula-3's back-end of SWIG the value of an identifier. This is necessary in the cases where it was defined by a non-trivial C expression. This feature is used by the <code>-generateconst</code> <a href="#">option</a> . In future it may be generalized to other kind of values such as strings.

### 20.5.2 Pragmas

Pragma	Example	Description
unsafe	<code>%pragma(modula3) unsafe="true";</code>	Mark the raw interface modules as UNSAFE. This will be necessary in many cases.
library	<code>%pragma(modula3) library="m3fftw";</code>	Specifies the library name for the wrapper library to be created. It should be distinct from the name of the library to be wrapped.

## 20.6 Remarks

- The Modula-3 part of SWIG doesn't try to generate nicely formatted code. Use `m3pp` to postprocess the Modula files, it does a very good job here.

## 21 SWIG and MzScheme

- [Creating native MzScheme structures](#)

This section contains information on SWIG's support of MzScheme.

### 21.1 Creating native MzScheme structures

Example interface file:

```
/* define a macro for the struct creation */
#define handle_ptr(TYPE,NAME)
%typemap(mzscheme,argout) TYPE *NAME{
    Scheme_Object *o = SWIG_NewStructFromPtr($1, $*1_mangle);
    SWIG_APPEND_VALUE(o);
}

%typemap(mzscheme,in,numinputs=0) TYPE *NAME (TYPE temp) {
    $1 = &temp;
}
#undef handle_ptr

/* setup the typemaps for the pointer to an output parameter cntrs */
handle_ptr(struct diag_cntrs, cntrs);
```

Then in scheme, you can use regular struct access procedures like

```
; suppose a function created a struct foo as
; (define foo (make-diag-cntrs (#x1 #x2 #x3) (make-inspector))
; Then you can do
(format "0x~x" (diag-cntrs-field1 foo))
(format "0x~x" (diag-cntrs-field2 foo))
;etc...
```

That's pretty much it. It works with nested structs as well.

## 22 SWIG and Ocaml

- [Preliminaries](#)
  - ◆ [Running SWIG](#)
  - ◆ [Compiling the code](#)
  - ◆ [The camlp4 module](#)
  - ◆ [Using your module](#)
  - ◆ [Compilation problems and compiling with C++](#)
- [The low-level Ocaml/C interface](#)
  - ◆ [The generated module](#)
  - ◆ [Enums](#)
    - ◇ [Enum typing in Ocaml](#)
  - ◆ [Arrays](#)
    - ◇ [Simple types of bounded arrays](#)
    - ◇ [Complex and unbounded arrays](#)
    - ◇ [Using an object](#)
    - ◇ [Example typemap for a function taking float \\* and int](#)
  - ◆ [C++ Classes](#)
    - ◇ [STL vector and string Example](#)
    - ◇ [C++ Class Example](#)
    - ◇ [Compiling the example](#)
    - ◇ [Sample Session](#)
  - ◆ [Director Classes](#)
    - ◇ [Director Introduction](#)
    - ◇ [Overriding Methods in Ocaml](#)
    - ◇ [Director Usage Example](#)
    - ◇ [Creating director objects](#)
    - ◇ [Typemaps for directors, directorin, directorout, directorargout](#)
    - ◇ [directorin typemap](#)
    - ◇ [directorout typemap](#)
    - ◇ [directorargout typemap](#)
  - ◆ [Exceptions](#)

This chapter describes SWIG's support of Ocaml. Ocaml is a relatively recent addition to the ML family, and is a recent addition to SWIG. It's the second compiled, typed language to be added. Ocaml has widely acknowledged benefits for engineers, mostly derived from a sophisticated type system, compile-time checking which eliminates several classes of common programming errors, and good native performance. While all of this is wonderful, there are well-written C and C++ libraries that Ocaml users will want to take advantage of as part of their arsenal (such as SSL and gdbm), as well as their own mature C and C++ code. SWIG allows this code to be used in a natural, type-safe way with Ocaml, by providing the necessary, but repetetive glue code which creates and uses Ocaml values to communicate with C and C++ code. In addition, SWIG also produces the needed Ocaml source that binds variants, functions, classes, etc.

If you're not familiar with the Objective Caml language, you can visit [The Ocaml Website](#).

### 22.1 Preliminaries

SWIG 1.3 works with Ocaml 3.04 and above. Given the choice, you should use the latest stable release. The SWIG Ocaml module has been tested on Linux (x86,PPC,Sparc) and Cygwin on Windows. The best way to determine whether your system will work is to compile the examples and test-suite which come with SWIG. You can do this by running `make check` from the SWIG root directory after installing SWIG. The Ocaml module has been tested using the system's dynamic linking (the usual `-lxxx` against `libxxx.so`, as well as with Gerd Stolpmann's [DL package](#) . The `ocaml_dynamic` and `ocaml_dynamic_cpp` targets in the file `Examples/Makefile` illustrate how to compile and link SWIG modules that will be loaded dynamically. This has only been tested on Linux so far.

## 22.1.1 Running SWIG

The basics of getting a SWIG Ocaml module up and running can be seen from one of SWIG's example Makefiles, but is also described here. To build an Ocaml module, run SWIG using the `-ocaml` option.

```
%swig -ocaml example.i
```

This will produce 3 files. The file `example_wrap.c` contains all of the C code needed to build an Ocaml module. To build the module, you will compile the file `example_wrap.c` with `ocamlc` or `ocamlopt` to create the needed `.o` file. You will need to compile the resulting `.ml` and `.mli` files as well, and do the final link with `-custom` (not needed for native link).

## 22.1.2 Compiling the code

The O'Caml SWIG module now requires you to compile a module (`Swig`) separately. In addition to aggregating common SWIG functionality, the `Swig` module contains the data structure that represents C/C++ values. This allows easier data sharing between modules if two or more are combined, because the type of each SWIG'ed module's `c_obj` is derived from `Swig.c_obj_t`. This also allows SWIG to acquire new conversions painlessly, as well as giving the user more freedom with respect to custom typing. Use `ocamlc` or `ocamlopt` to compile your SWIG interface like:

```
% swig -ocaml -co swig.mli ; swig -ocaml co swig.ml
% ocamlc -c swig.mli ; ocamlc -c swig.ml
% ocamlc -c -ccopt "-I/usr/include/foo" example_wrap.c
% ocamlc -c example.mli
% ocamlc -c example.ml
```

`ocamlc` is aware of `.c` files and knows how to handle them. Unfortunately, it does not know about `.cxx`, `.cc`, or `.cpp` files, so when SWIG is invoked in C++ mode, you must:

```
% cp example_wrap.cxx example_wrap.cxx.c
% ocamlc -c ... -ccopt -xc++ example_wrap.cxx.c
% ...
```

## 22.1.3 The camlp4 module

The `camlp4` module (`swigp4.ml` → `swigp4.cmo`) contains a simple rewriter which makes C++ code blend more seamlessly with objective caml code. It's use is optional, but encouraged. The source file is included in the `Lib/ocaml` directory of the SWIG source distribution. You can checkout this file with `"swig -ocaml -co swigp4.ml"`. You should compile the file with `"ocamlc -I `camlp4 -where` -pp 'camlp4o pa_extend.cmo q_MLast.cmo' -c swigp4.ml"`

The basic principle of the module is to recognize certain non-caml expressions and convert them for use with C++ code as interfaced by SWIG. The `camlp4` module is written to work with generated SWIG interfaces, and probably isn't great to use with anything else.

Here are the main rewriting rules:

Input	Rewritten to
<code>f( ... )</code> as in <code>atoi("0")</code> or <code>_exit(0)</code>	<code>f(C_list [ ... ])</code> as in <code>atoi (C_list [ C_string "0" ])</code> or <code>_exit (C_list [ C_int 0 ])</code>
<code>object → method ( ... )</code>	<code>(invoke object) "method" (C_list [ ... ])</code>
<code>object</code> <i>binop</i> argument as in <code>a += b</code>	<code>(invoke object) "+="</code> argument as in

	(invoke a) "+=" b
<b>Note that because camlp4 always recognizes &lt;&lt; and &gt;&gt;, they are replaced by lsl and lsr in operator names.</b>	
'unop object as in '! a	(invoke a) "!" C_void
<b>Smart pointer access like this</b> object '-> method ( args )	(invoke (invoke object "->" C_void))
<b>Invoke syntax</b> object . '( ... )	(invoke object) "()" (C_list [ ... ])
<b>Array syntax</b> object '[ 10 ]	(invoke object) "[]" (C_int 10)
<b>Assignment syntax</b> let a = '10 and b = ""foo" and c = '1.0 and d = 'true	let a = C_int 10 and b = C_string "foo" and c = C_double 1.0 and d = C_bool true
<b>Cast syntax</b> let a = _atoi ("2") as int let b = (getenv "PATH") to string This works for int, string, float, bool	let a = get_int (_atoi (C_string "2")) let b = C_string (getenv "PATH")

### 22.1.4 Using your module

You can test-drive your module by building a toplevel ocaml interpreter. Consult the ocaml manual for details.

When linking any ocaml bytecode with your module, use the `-custom` option to build your functions into the primitive list. This option is not needed when you build native code.

### 22.1.5 Compilation problems and compiling with C++

As mentioned above, `.cxx` files need special handling to be compiled with `ocamlc`. Other than that, C code that uses `class` as a non-keyword, and C code that is too liberal with pointer types may not compile under the C++ compiler. Most code meant to be compiled as C++ will not have problems.

## 22.2 The low-level Ocaml/C interface

In order to provide access to overloaded functions, and provide sensible outputs from them, all C entites are represented as members of the `c_obj` type:

In the code as seen by the typemap writer, there is a value, `swig_result`, that always contains the current return data. It is a list, and must be appended with the `caml_list_append` function, or with functions and macros provided by objective caml.

```

type c_obj =
  C_void
  | C_bool of bool
  | C_char of char
  | C_uchar of char
  | C_short of int
  | C_ushort of int
  | C_int of int
  | C_uint of int32
  | C_int32 of int32
  | C_int64 of int64
  | C_float of float
  | C_double of float
  | C_ptr of int64 * int64
  | C_array of c_obj array
  | C_list of c_obj list

```

```
| C_obj of (string -> c_obj -> c_obj)
| C_string of string
| C_enum of c_enum_t
```

A few functions exist which generate and return these:

- `caml_ptr_val` receives a `c_obj` and returns a `void *`. This should be used for all pointer purposes.
- `caml_long_val` receives a `c_obj` and returns a `long`. This should be used for most integral purposes.
- `caml_val_ptr` receives a `void *` and returns a `c_obj`.
- `caml_val_bool` receives a C int and returns a `c_obj` representing its bool value.
- `caml_val_(u)?(char|short|int|long|float|double)` receives an appropriate C value and returns a `c_obj` representing it.
- `caml_val_string` receives a `char *` and returns a string value.
- `caml_val_string_len` receives a `char *` and a length and returns a string value.
- `caml_val_obj` receives a `void *` and an object type and returns a `C_obj`, which contains a closure giving method access.

Because of this style, a typemap can return any kind of value it wants from a function. This enables out typemaps and inout typemaps to work well. The one thing to remember about outputting values is that you must append them to the return list with `swig_result = caml_list_append(swig_result,v)`.

This function will return a new list that has your element appended. Upon return to caml space, the `fnhelper` function beautifies the result. A list containing a single item degrades to only that item (i.e. `[ C_int 3 ] -> C_int 3`), and a list containing more than one item is wrapped in `C_list` (i.e. `[ C_char 'a'; C_char 'b' -> C_list [ C_char 'a'; C_char b ]`). This is in order to make return values easier to handle when functions have only one return value, such as constructors, and operators. In addition, string, pointer, and object values are interchangeable with respect to `caml_ptr_val`, so you can allocate memory as caml strings and still use the resulting pointers for C purposes, even using them to construct simple objects on. Note, though, that foreign C++ code does not respect the garbage collector, although the SWIG interface does.

The wild card type that you can use in lots of different ways is `C_obj`. It allows you to wrap any type of thing you like as an object using the same mechanism that the `ocaml` module does. When evaluated in `caml_ptr_val`, the returned value is the result of a call to the object's `"&"` operator, taken as a pointer.

You should only construct values using objective caml, or using the functions `caml_val_*` functions provided as static functions to a SWIG ocaml module, as well as the `caml_list_*` functions. These functions provide everything a typemap needs to produce values. In addition, value items pass through directly, but you must make your own type signature for a function that uses value in this way.

### 22.2.1 The generated module

The SWIG `%module` directive specifies the name of the Ocaml module to be generated. If you specified ``%module example'`, then your Ocaml code will be accessible in the module `Example`. The module name is always capitalized as is the ocaml convention. Note that you must not use any Ocaml keyword to name your module. Remember that the keywords are not the same as the C++ ones.

You can introduce extra code into the output wherever you like with SWIG. These are the places you can introduce code:

"header"	This code is inserted near the beginning of the C wrapper file, before any function definitions.
"wrapper"	This code is inserted in the function definition section.
"runtime"	This code is inserted near the end of the C wrapper file.
"mli"	This code is inserted into the caml interface file. Special signatures should be inserted here.
"ml"	This code is inserted in the caml code defining the interface to your C code. Special caml code, as well as any initialization which should run when the module is loaded may be inserted here.
"classtemplate"	The "classtemplate" place is special because it describes the output SWIG will generate for class definitions.

## 22.2.2 Enums

SWIG will wrap enumerations as polymorphic variants in the output Ocaml code, as above in `C_enum`. In order to support all C++-style uses of enums, the function `int_to_enum` and `enum_to_int` are provided for ocaml code to produce and consume these values as integers. Other than that, correct uses of enums will not have a problem. Since enum labels may overlap between enums, the `enum_to_int` and `int_to_enum` functions take an enum type label as an argument. Example:

```
%module enum_test
%{
enum c_enum_type { a = 1, b, c = 4, d = 8 };
%}
enum c_enum_type { a = 1, b, c = 4, d = 8 };
```

The output mli contains:

```
type c_enum_type = [
  `unknown
| `c_enum_type
]
type c_enum_tag = [
  `int of int
| `a
| `b
| `c
| `d
]
val int_to_enum c_enum_type -> int -> c_obj
val enum_to_int c_enum_type -> c_obj -> c_obj
```

So it's possible to do this:

```
bash-2.05a$ ocamlmktop -custom enum_test_wrap.o enum_test.cmo -o enum_test_top
bash-2.05a$ ./enum_test_top
Objective Caml version 3.04

# open Enum_test ;;
# let x = C_enum `a ;;
val x : Enum_test.c_obj = C_enum `a
# enum_to_int `c_enum_type x ;;
- : Enum_test.c_obj = C_int 1
# int_to_enum `c_enum_type 4 ;;
- : Enum_test.c_obj = C_enum `c
```

### 22.2.2.1 Enum typing in Ocaml

The ocaml SWIG module now has support for loading and using multiple SWIG modules at the same time. This enhances modularity, but presents problems when used with a language which assumes that each module's types are complete at compile time. In order to achieve total soundness enum types are now isolated per-module. The type issue matters when values are shared between functions imported from different modules. You must convert values to master values using the `swig_val` function before sharing them with another module.

## 22.2.3 Arrays

### 22.2.3.1 Simple types of bounded arrays

SWIG has support for array types, but you generally will need to provide a typemap to handle them. You can currently roll your own, or expand some of the macros provided (but not included by default) with the SWIG distribution.

By including "carray.i", you will get access to some macros that help you create typemaps for array types fairly easily.

`%make_simple_array_typemap` is the easiest way to get access to arrays of simple types with known bounds in your code, but this only works for arrays whose bounds are completely specified.

### 22.2.3.2 Complex and unbounded arrays

Unfortunately, unbounded arrays and pointers can't be handled in a completely general way by SWIG, because the end-condition of such an array can't be predicted. In some cases, it will be by consent (e.g. an array of four or more chars), sometimes by explicit length (`char *buffer, int len`), and sometimes by sentinel value (0,-1,etc.). SWIG can't predict which of these methods will be used in the array, so you have to specify it for yourself in the form of a typemap.

### 22.2.3.3 Using an object

It's possible to use C++ to your advantage by creating a simple object that provides access to your array. This may be more desirable in some cases, since the object can provide bounds checking, etc., that prevents crashes.

Consider writing an object when the ending condition of your array is complex, such as using a required sentinel, etc.

### 22.2.3.4 Example typemap for a function taking float \* and int

This is a simple example in typemap for an array of float, where the length of the array is specified as an extra parameter. Other such typemaps will work similarly. In the example, the function `printfloats` is called with a float array, and specified length. The actual length reported in the `len` argument is the length of the array passed from ocaml, making passing an array into this type of function convenient.

tarray.i
<pre>%module tarray %{ #include &lt;stdio.h&gt;  void printfloats( float *tab, int len ) {     int i;      for( i = 0; i &lt; len; i++ ) {         printf( "%f ", tab[i] );     }      printf( "\n" ); } %}  %typemap(in) (float *tab, int len) {     int i;     /* \$*1_type */     \$2 = caml_array_len(\$input);     \$1 = (\$*1_type *)malloc( \$2 * sizeof( float ) );     for( i = 0; i &lt; \$2; i++ ) {         \$1[i] = caml_double_val(caml_array_nth(\$input,i));     } }  void printfloats( float *tab, int len );</pre>
Sample Run
<pre># open Tarray ;; # _printfloats (C_array [  C_double 1.0 ; C_double 3.0 ; C_double 5.6666  ]) ;; 1.000000 3.000000 5.666600 - : Tarray.c_obj = C_void</pre>



## 22.2.4 C++ Classes

C++ classes, along with structs and unions are represented by `C_obj` (`string -> c_obj -> c_obj`) wrapped closures. These objects contain a method list, and a type, which allow them to be used like C++ objects. When passed into typemaps that use pointers, they degrade to pointers through their `"&"` method. Every method an object has is represented as a string in the object's method table, and each method table exists in memory only once. In addition to any other operators an object might have, certain builtin ones are provided by SWIG: (all of these take no arguments (`C_void`))

<code>"~"</code>	Delete this object
<code>"&amp;"</code>	Return an ordinary <code>C_ptr</code> value representing this object's address
<code>"sizeof"</code>	If enabled with ( <code>"sizeof"="1"</code> ) on the module node, return the object's size in char.
<code>":methods"</code>	Returns a list of strings containing the names of the methods this object contains
<code>":classof"</code>	Returns the name of the class this object belongs to.
<code>":parents"</code>	Returns a list of all direct parent classes which have been wrapped by SWIG.
<code>"::[parent-class]"</code>	Returns a view of the object as the indicated parent class. This is mainly used internally by the SWIG module, but may be useful to client programs.
<code>"[member-variable]"</code>	Each member variable is wrapped as a method with an optional parameter. Called with one argument, the member variable is set to the value of the argument. With zero arguments, the value is returned.

Note that this string belongs to the wrapper object, and not the underlying pointer, so using `create_[x]_from_ptr` alters the returned value for the same object.

### 22.2.4.1 STL vector and string Example

Standard typemaps are now provided for STL vector and string. More are in the works. STL strings are passed just like normal strings, and returned as strings. STL string references don't mutate the original string, (which might be surprising), because Ocaml strings are mutable but have fixed length. Instead, use multiple returns, as in the `argout_ref` example.

example.i
<pre>%module example %{ #include "example.h" %}  #include stl.i  namespace std {     %template(StringVector) std::vector &lt; string &gt;; };  #include example.h</pre>
<i>This example is in Examples/ocaml/stl</i>

Since there's a makefile in that directory, the example is easy to build.

Here's a sample transcript of an interactive session using a string vector after making a toplevel (`make toplevel`). This example uses the `camlp4` module.

```
bash-2.05a$ ./example_top
Objective Caml version 3.06

Camlp4 Parsing version 3.06

# open Swig ;;
# open Example ;;
# let x = new_StringVector '() ;;
val x : Example.c_obj = C_obj <fun>
# x -> ":methods" () ;;
```

```

- : Example.c_obj =
C_list
[C_string "nop"; C_string "size"; C_string "empty"; C_string "clear";
C_string "push_back"; C_string "["; C_string "="; C_string "set";
C_string "~"; C_string "&"; C_string ":parents"; C_string ":classof";
C_string ":methods"]
# x -> push_back ("foo") ;;
- : Example.c_obj = C_void
# x -> push_back ("bar") ;;
- : Example.c_obj = C_void
# x -> push_back ("baz") ;;
- : Example.c_obj = C_void
# x '[1] ;;
- : Example.c_obj = C_string "bar"
# x -> set (1,"spam") ;;
- : Example.c_obj = C_void
# x '[1] ;;
- : Example.c_obj = C_string "spam"
# for i = 0 to (x -> size() as int) - 1 do
  print_endline ((x '[i to int]) as string)
done ;;
foo
bar
baz
- : unit = ()
#

```

#### 22.2.4.2 C++ Class Example

Here's a simple example using Trolltech's Qt Library:

qt.i
<pre> %module qt %{ #include &lt;qapplication.h&gt; #include &lt;qpushbutton.h&gt; %} class QApplication { public:     QApplication( int argc, char **argv );     void setMainWidget( QWidget *widget );     void exec(); };  class QPushButton { public:     QPushButton( char *str, QWidget *w );     void resize( int x, int y );     void show(); }; </pre>

#### 22.2.4.3 Compiling the example

```

bash-2.05a$ QTPATH=/your/qt/path
bash-2.05a$ for file in swig.mli swig.ml swigp4.ml ; do swig -ocaml -co $file ; done
bash-2.05a$ ocamlc -c swig.mli ; ocamlc -c swig.ml
bash-2.05a$ ocamlc -I `camlp4 -where` -pp "camlp4o pa_extend.cmo q_MLast.cmo" -c swigp4.ml
bash-2.05a$ swig -ocaml -c++ -I$QTPATH/include qt.i
bash-2.05a$ mv qt_wrap.cxx qt_wrap.c
bash-2.05a$ ocamlc -c -ccopt -xc++ -ccopt -g -g -ccopt -I$QTPATH/include qt_wrap.c
bash-2.05a$ ocamlc -c qt.mli
bash-2.05a$ ocamlc -c qt.ml
bash-2.05a$ ocamlmktop -custom swig.cmo -I `camlp4 -where` \
    camlp4o.cma swigp4.cmo qt_wrap.o qt.cmo -o qt_top -cclib \
    -L$QTPATH/lib -cclib -lqt

```

### 22.2.4.4 Sample Session

```
bash-2.05a$ ./qt_top
Objective Caml version 3.06

Camlp4 Parsing version 3.06

# open Swig ;;
# open Qt ;;
# let a = new_QApplication '(0,0) ;;
val a : Qt.c_obj = C_obj <fun>
# let hello = new_QPushButton '("hi",0) ;;
val hello : Qt.c_obj = C_obj <fun>
# hello -> resize (100,30) ;;
- : Qt.c_obj = C_void
# hello -> show () ;;
- : Qt.c_obj = C_void
# a -> exec () ;;
```

Assuming you have a working installation of QT, you will see a window containing the string "hi" in a button.

## 22.2.5 Director Classes

### 22.2.5.1 Director Introduction

Director classes are classes which allow Ocaml code to override the public methods of a C++ object. This facility allows the user to use C++ libraries that require a derived class to provide application specific functionality in the context of an application or utility framework.

You can turn on director classes by using an optional module argument like this:

```
%module(directors="1")

...

// Turn on the director class for a specific class like this:
%feature("director")
class foo {
  ...
};
```

### 22.2.5.2 Overriding Methods in Ocaml

Because the Ocaml language module treats C++ method calls as calls to a certain function, all you need to do is to define the function that will handle the method calls in terms of the public methods of the object, and any other relevant information. The function `new_derived_object` uses a stub class to call your methods in place of the ones provided by the underlying implementation. The object you receive is the underlying object, so you are free to call any methods you want from within your derived method. Note that calls to the underlying object do not invoke Ocaml code. You need to handle that yourself.

`new_derived_object` receives your function, the function that creates the underlying object, and any constructor arguments, and provides an object that you can use in any usual way. When C++ code calls one of the object's methods, the object invokes the Ocaml function as if it had been invoked from Ocaml, allowing any method definitions to override the C++ ones.

In this example, I'll examine the objective caml code involved in providing an overloaded class. This example is contained in `Examples/ocaml/shapes`.

### 22.2.5.3 Director Usage Example

example_prog.ml
open Swig

```

open Example

...

let triangle_class pts ob meth args =
  match meth with
  | "cover" ->
    (match args with
     | C_list [ x_arg ; y_arg ] ->
       let xa = x_arg as float
       and ya = y_arg as float in
       (point_in_triangle pts xa ya) to bool
     | _ -> raise (Failure "cover needs two double arguments. "))
  | _ -> (invoke ob) meth args ;;

let triangle =
  new_derived_object
  new_shape
  (triangle_class ((0.0,0.0),(0.5,1.0),(1.0,0.0)))
  '() ;;

let _ = _draw_shape_coverage '(triangle, C_int 60, C_int 20) ;;

```

This is the meat of what you need to do. The actual "class" definition containing the overloaded method is defined in the function `triangle_class`. This is a lot like the class definitions emitted by SWIG, if you look at `example.ml`, which is generated when SWIG consumes `example.i`. Basically, you are given the arguments as a `c_obj` and the method name as a string, and you must intercept the method you are interested in and provide whatever return value you need. Bear in mind that the underlying C++ code needs the right return type, or an exception will be thrown. This exception will generally be `Failure`, or `NotObject`. You must call other ocaml methods that you rely on yourself. Due to the way directors are implemented, method calls on your object from within ocaml code will always invoke C++ methods even if they are overridden in ocaml.

In the example, the `draw_shape_coverage` function plots the indicated number of points as either covered (x) or uncovered ( ) between 0 and 1 on the X and Y axes. Your shape implementation can provide any coverage map it likes, as long as it responds to the "cover" method call with a boolean return (the underlying method returns `bool`). This might allow a tricky shape implementation, such as a boolean combination, to be expressed in a more effortless style in ocaml, while leaving the "engine" part of the program in C++.

#### 22.2.5.4 Creating director objects

The definition of the actual object `triangle` can be described this way:

```

let triangle =
  new_derived_object
  new_shape
  (triangle_class ((0.0,0.0),(0.5,1.0),(1.0,0.0)))
  '()

```

The first argument to `new_derived_object`, `new_shape` is the method which returns a shape instance. This function will be invoked with the third argument will be appended to the argument list `[ C_void ]`. In the example, the actual argument list is sent as `(C_list [ C_void ; C_void ])`. The augmented constructor for a director class needs the first argument to determine whether it is being constructed as a derived object, or as an object of the indicated type only (in this case `shape`). The Second argument is a closure that will be added to the final `C_obj`.

The actual object passed to the `self` parameter of the director object will be a `C_director_core`, containing a `c_obj` option ref and a `c_obj`. The `c_obj` provided is the same object that will be returned from `new_derived` object, that is, the object exposing the overridden methods. The other part is an option ref that will have its value extracted before becoming the `ob` parameter of your class closure. This ref will contain `None` if the C++ object underlying is ever destroyed, and will consequently trigger an exception when any method is called on the object after that point (the actual raise is from an inner function used by `new_derived_object`, and throws `NotObject`). This prevents a deleted C++ object from causing a core dump, as long as the object is destroyed properly.

### 22.2.5.5 Typemaps for `directors`, `directorin`, `directorout`, `directorargout`

Special typemaps exist for use with directors, the `directorin`, `directorout`, `directorargout` are used in place of `in`, `out`, `argout` typemaps, except that their direction is reversed. They provide for you to provide argout values, as well as a function return value in the same way you provide function arguments, and to receive arguments the same way you normally receive function returns.

#### 22.2.5.6 `directorin` typemap

The `directorin` typemap is used when you will receive arguments from a call made by C++ code to you, therefore, values will be translated from C++ to ocaml. You must provide some valid `C_obj` value. This is the value your ocaml code receives when you are called. In general, a simple `directorin` typemap can use the same body as a simple `out` typemap.

#### 22.2.5.7 `directorout` typemap

The `directorout` typemap is used when you will send an argument from your code back to the C++ caller. That is; `directorout` specifies a function return conversion. You can usually use the same body as an `in` typemap for the same type, except when there are special requirements for object ownership, etc.

#### 22.2.5.8 `directorargout` typemap

C++ allows function arguments which are by pointer (\*) and by reference (&) to receive a value from the called function, as well as sending one there. Sometimes, this is the main purpose of the argument given. `directorargout` typemaps allow your caml code to emulate this by specifying additional return values to be put into the output parameters. The SWIG ocaml module is a bit loose in order to make code easier to write. In this case, your return to the caller must be a list containing the normal function return first, followed by any argout values in order. These argout values will be taken from the list and assigned to the values to be returned to C++ through `directorargout` typemaps. In the event that you don't specify all of the necessary values, integral values will read zero, and struct or object returns have undefined results.

### 22.2.6 Exceptions

Catching exceptions is now supported using SWIG's `%exception` feature. A simple but not too useful example is provided by the `throw_exception` testcase in `Examples/test-suite`. You can provide your own exceptions, too.

## 23 SWIG and Perl5

- [Overview](#)
- [Preliminaries](#)
  - ◆ [Getting the right header files](#)
  - ◆ [Compiling a dynamic module](#)
  - ◆ [Building a dynamic module with MakeMaker](#)
  - ◆ [Building a static version of Perl](#)
  - ◆ [Using the module](#)
  - ◆ [Compilation problems and compiling with C++](#)
  - ◆ [Compiling for 64-bit platforms](#)
- [Building Perl Extensions under Windows](#)
  - ◆ [Running SWIG from Developer Studio](#)
  - ◆ [Using other compilers](#)
- [The low-level interface](#)
  - ◆ [Functions](#)
  - ◆ [Global variables](#)
  - ◆ [Constants](#)
  - ◆ [Pointers](#)
  - ◆ [Structures](#)
  - ◆ [C++ classes](#)
  - ◆ [C++ classes and type-checking](#)
  - ◆ [C++ overloaded functions](#)
  - ◆ [Operators](#)
  - ◆ [Modules and packages](#)
- [Input and output parameters](#)
- [Exception handling](#)
- [Remapping datatypes with typemaps](#)
  - ◆ [A simple typemap example](#)
  - ◆ [Perl5 typemaps](#)
  - ◆ [Typemap variables](#)
  - ◆ [Useful functions](#)
- [Typemap Examples](#)
  - ◆ [Converting a Perl5 array to a char \\*\\*](#)
  - ◆ [Return values](#)
  - ◆ [Returning values from arguments](#)
  - ◆ [Accessing array structure members](#)
  - ◆ [Turning Perl references into C pointers](#)
  - ◆ [Pointer handling](#)
- [Proxy classes](#)
  - ◆ [Preliminaries](#)
  - ◆ [Structure and class wrappers](#)
  - ◆ [Object Ownership](#)
  - ◆ [Nested Objects](#)
  - ◆ [Proxy Functions](#)
  - ◆ [Inheritance](#)
  - ◆ [Modifying the proxy methods](#)

**Caution:** This chapter is under repair!

This chapter describes SWIG's support of Perl5. Although the Perl5 module is one of the earliest SWIG modules, it has continued to evolve and has been improved greatly with the help of SWIG users. For the best results, it is recommended that SWIG be used with Perl5.003 or later. Earlier versions are problematic and SWIG generated extensions may not compile or run correctly.

## 23.1 Overview

To build Perl extension modules, SWIG uses a layered approach. At the lowest level, simple procedural wrappers are generated for functions, classes, methods, and other declarations in the input file. Then, for structures and classes, an optional collection of Perl proxy classes can be generated in order to provide a more natural object oriented Perl interface. These proxy classes simply build upon the low-level interface.

In describing the Perl interface, this chapter begins by covering the essentials. First, the problem of configuration, compiling, and installing Perl modules is discussed. Next, the low-level procedural interface is presented. Finally, proxy classes are described. Advanced customization features, typemaps, and other options are found near the end of the chapter.

## 23.2 Preliminaries

To build a Perl5 module, run Swig using the `-perl` option as follows :

```
swig -perl example.i
```

This produces two files. The first file, `example_wrap.c` contains all of the C code needed to build a Perl5 module. The second file, `example.pm` contains supporting Perl code needed to properly load the module.

To build the module, you will need to compile the file `example_wrap.c` and link it with the rest of your program.

### 23.2.1 Getting the right header files

In order to compile, SWIG extensions need the following Perl5 header files :

```
#include "Extern.h"
#include "perl.h"
#include "XSUB.h"
```

These are typically located in a directory like this

```
/usr/lib/perl5/5.00503/i386-linux/CORE
```

The SWIG configuration script automatically tries to locate this directory so that it can compile examples. However, if you need to find out where the directory is loaded, an easy way to find out is to run Perl itself.

```
% perl -e 'use Config; print $Config{archlib};'
/usr/lib/perl5/5.00503/i386-linux
```

### 23.2.2 Compiling a dynamic module

The preferred approach to building an extension module is to compile it into a shared object file or DLL. To do this, you will need to compile your program using commands like this (shown for Linux):

```
$ swig -perl example.i
% gcc example.c
% gcc -c example_wrap.c -I/usr/lib/perl5/5.00503/i386-linux/CORE -Dbool=char
% gcc -shared example.o example_wrap.o -o example.so
```

The exact compiler options vary from platform to platform. SWIG tries to guess the right options when it is installed. Therefore, you may want to start with one of the examples in the `SWIG/Examples/perl5` directory. If that doesn't work, you will need to read the man-pages for your compiler and linker to get the right set of options. You might also check the [SWIG Wiki](#) for additional information.

When linking the module, the name of the shared object file must match the module name used in the SWIG interface file. If you used `%module example`, then the target should be named `example.so`, `example.sl`, or the appropriate dynamic module name on your system.

### 23.2.3 Building a dynamic module with MakeMaker

It is also possible to use Perl to build dynamically loadable modules for you using the MakeMaker utility. To do this, write a Perl script such as the following :

```
# File : Makefile.PL
use ExtUtils::MakeMaker;
WriteMakefile(
    'NAME'      => 'example',           # Name of package
    'LIBS'      => ['-lm'],             # Name of custom libraries
    'OBJECT'    => 'example.o example_wrap.o' # Object files
);
```

Now, to build a module, simply follow these steps :

```
% perl Makefile.PL
% make
% make install
```

If you are planning to distribute a SWIG-generated module, this is the preferred approach to compilation. More information about MakeMaker can be found in "Programming Perl, 2nd ed." by Larry Wall, Tom Christiansen, and Randal Schwartz.

### 23.2.4 Building a static version of Perl

If your machine does not support dynamic loading or if you've tried to use it without success, you can build a new version of the Perl interpreter with your SWIG extensions added to it. To build a static extension, you first need to invoke SWIG as follows :

```
% swig -perl -static example.i
```

By default SWIG includes code for dynamic loading, but the `-static` option takes it out.

Next, you will need to supply a `main()` function that initializes your extension and starts the Perl interpreter. While, this may sound daunting, SWIG can do this for you automatically as follows :

```
%module example

extern double My_variable;
extern int fact(int);

// Include code for rebuilding Perl
#include perlmain.i
```

The same thing can be accomplished by running SWIG as follows :

```
% swig -perl -static -lperlmain.i example.i
```

The `perlmain.i` file inserts Perl's `main()` function into the wrapper code and automatically initializes the SWIG generated module. If you just want to make a quick dirty module, this may be the easiest way. By default, the `perlmain.i` code does not initialize any other Perl extensions. If you need to use other packages, you will need to modify it appropriately. You can do this by just copying `perlmain.i` out of the SWIG library, placing it in your own directory, and modifying it to suit your purposes.

To build your new Perl executable, follow the exact same procedure as for a dynamic module, but change the link line to something like this:



```
% gcc example.o example_wrap.o -L/usr/lib/perl5/5.00503/i386-linux/CORE \
    -lperl -lsocket -lnsl -lm -o myperl
```

This will produce a new version of Perl called `myperl`. It should be functionality identical to Perl with your C/C++ extension added to it. Depending on your machine, you may need to link with additional libraries such as `-lsocket`, `-lnsl`, `-ldl`, etc.

### 23.2.5 Using the module

To use the module, simply use the Perl `use` statement. If all goes well, you will be able to do this:

```
$ perl
use example;
print example::fact(4), "\n";
24
```

A common error received by first-time users is the following:

```
use example;
Can't locate example.pm in @INC (@INC contains: /usr/lib/perl5/5.00503/i386-lin
ux /usr/lib/perl5/5.00503 /usr/lib/perl5/site_perl/5.005/i386-linux /usr/lib/pe
rl5/site_perl/5.005 .) at - line 1.
BEGIN failed--compilation aborted at - line 1.
```

This error is almost caused when the name of the shared object file you created doesn't match the module name you specified with the `%module` directive.

A somewhat related, but slightly different error is this:

```
use example;
Can't find 'boot_example' symbol in ./example.so
at - line 1
BEGIN failed--compilation aborted at - line 1.
```

This error is generated because Perl can't locate the module bootstrap function in the SWIG extension module. This could be caused by a mismatch between the module name and the shared library name. However, another possible cause is forgetting to link the SWIG-generated wrapper code with the rest of your application when you linked the extension module.

Another common error is the following:

```
use example;
Can't load './example.so' for module example: ./example.so:
undefined symbol: Foo at /usr/lib/perl5/5.00503/i386-linux/DynaLoader.pm line 169.

at - line 1
BEGIN failed--compilation aborted at - line 1.
```

This error usually indicates that you forgot to include some object files or libraries in the linking of the shared library file. Make sure you compile both the SWIG wrapper file and your original program into a shared library file. Make sure you pass all of the required libraries to the linker.

Sometimes unresolved symbols occur because a wrapper has been created for a function that doesn't actually exist in a library. This usually occurs when a header file includes a declaration for a function that was never actually implemented or it was removed from a library without updating the header file. To fix this, you can either edit the SWIG input file to remove the offending declaration or you can use the `%ignore` directive to ignore the declaration. Better yet, update the header file so that it doesn't have an undefined declaration.

Finally, suppose that your extension module is linked with another library like this:

```
$ gcc -shared example.o example_wrap.o -L/home/beazley/projects/lib -lfoo \
```

```
-o example.so
```

If the `foo` library is compiled as a shared library, you might get the following error when you try to use your module:

```
use example;
Can't load './example.so' for module example: libfoo.so: cannot open shared object file:
No such file or directory at /usr/lib/perl5/5.00503/i386-linux/DynaLoader.pm line 169.

at - line 1
BEGIN failed--compilation aborted at - line 1.
>>>
```

This error is generated because the dynamic linker can't locate the `libfoo.so` library. When shared libraries are loaded, the system normally only checks a few standard locations such as `/usr/lib` and `/usr/local/lib`. To get the loader to look in other locations, there are several things you can do. First, you can recompile your extension module with extra path information. For example, on Linux you can do this:

```
$ gcc -shared example.o example_wrap.o -L/home/beazley/projects/lib -lfoo \
-Xlinker -rpath /home/beazley/projects/lib \
-o example.so
```

Alternatively, you can set the `LD_LIBRARY_PATH` environment variable to include the directory with your shared libraries. If setting `LD_LIBRARY_PATH`, be aware that setting this variable can introduce a noticeable performance impact on all other applications that you run. To set it only for Perl, you might want to do this instead:

```
$ env LD_LIBRARY_PATH=/home/beazley/projects/lib perl
```

Finally, you can use a command such as `ldconfig` (Linux) or `crle` (Solaris) to add additional search paths to the default system configuration (this requires root access and you will need to read the man pages).

### 23.2.6 Compilation problems and compiling with C++

Compilation of C++ extensions has traditionally been a tricky problem. Since the Perl interpreter is written in C, you need to take steps to make sure C++ is properly initialized and that modules are compiled correctly.

On most machines, C++ extension modules should be linked using the C++ compiler. For example:

```
% swig -c++ -perl example.i
% g++ -c example.cxx
% g++ -c example_wrap.cxx -I/usr/lib/perl5/5.00503/i386-linux/CORE
% g++ -shared example.o example_wrap.o -o example.so
```

In addition to this, you may need to include additional library files to make it work. For example, if you are using the Sun C++ compiler on Solaris, you often need to add an extra library `-lCrun` like this:

```
% swig -c++ -perl example.i
% g++ -c example.cxx
% g++ -c example_wrap.cxx -I/usr/lib/perl5/5.00503/i386-linux/CORE
% g++ -shared example.o example_wrap.o -o example.so -lCrun
```

Of course, the names of the extra libraries are completely non-portable—you will probably need to do some experimentation.

Another possible compile problem comes from recent versions of Perl (5.8.0) and the GNU tools. If you see errors having to do with `_crypt_struct`, that means `_GNU_SOURCE` is not defined and it needs to be. So you should compile the wrapper like:

```
% g++ -c example_wrap.cxx -I/usr/lib/perl5/5.8.0/CORE -D_GNU_SOURCE
```

`-D_GNU_SOURCE` is also included in the Perl `ccflags`, which can be found by running

```
% perl -e 'use Config; print $Config{ccflags};'
```

So you could also compile the wrapper like

```
% g++ -c example_wrap.cxx -I/usr/lib/perl/5.8.0/CORE \
`perl -e 'use Config; print $Config{ccflags}``
```

Sometimes people have suggested that it is necessary to relink the Perl interpreter using the C++ compiler to make C++ extension modules work. In the experience of this author, this has never actually appeared to be necessary on most platforms. Relinking the interpreter with C++ really only includes the special run-time libraries described above---as long as you link your extension modules with these libraries, it should not be necessary to rebuild Perl.

If you aren't entirely sure about the linking of a C++ extension, you might look at an existing C++ program. On many Unix machines, the `ldd` command will list library dependencies. This should give you some clues about what you might have to include when you link your extension module. For example, notice the first line of output here:

```
$ ldd swig
      libstdc++-libc6.1-1.so.2 => /usr/lib/libstdc++-libc6.1-1.so.2 (0x40019000)
      libm.so.6 => /lib/libm.so.6 (0x4005b000)
      libc.so.6 => /lib/libc.so.6 (0x40077000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

If linking wasn't enough of a problem, another major complication of C++ is that it does not define any sort of standard for binary linking of libraries. This means that C++ code compiled by different compilers will not link together properly as libraries nor is the memory layout of classes and data structures implemented in any kind of portable manner. In a monolithic C++ program, this problem may be unnoticed. However, in Perl, it is possible for different extension modules to be compiled with different C++ compilers. As long as these modules are self-contained, this probably won't matter. However, if these modules start sharing data, you will need to take steps to avoid segmentation faults and other erratic program behavior. Also, be aware that certain C++ features, especially RTTI, can behave strangely when working with multiple modules.

It should be noted that you may get a lot of error messages about the `'bool'` datatype when compiling a C++ Perl module. If you experience this problem, you can try the following :

- Use `-DHAS_BOOL` when compiling the SWIG wrapper code
- Or use `-Dbool=char` when compiling.

Finally, recent versions of Perl (5.8.0) have namespace conflict problems. Perl defines a bunch of short macros to make the Perl API function names shorter. For example, in `/usr/lib/perl/5.8.0/CORE/embed.h` there is a line:

```
#define do_open Perl_do_open
```

The problem is, in the `<iostream>` header from GNU `libstdc++v3` there is a private function named `do_open`. If `<iostream>` is included after the perl headers, then the Perl macro causes the `iostream` `do_open` to be renamed, which causes compile errors. Hopefully in the future Perl will support a `PERL_NO_SHORT_NAMES` flag, but for now the only solution is to undef the macros that conflict. `Lib/perl5/noembed.h` in the SWIG source has a list of macros that are known to conflict with either standard headers or other headers. But if you get macro type conflicts from other macros not included in `Lib/perl5/noembed.h` while compiling the wrapper, you will have to find the macro that conflicts and add an `#undef` into the `.i` file. Please report any conflicting macros you find to [swig mailing list](#).

### 23.2.7 Compiling for 64-bit platforms

On platforms that support 64-bit applications (Solaris, Irix, etc.), special care is required when building extension modules. On these machines, 64-bit applications are compiled and linked using a different set of compiler/linker options. In addition, it is not generally possible to mix 32-bit and 64-bit code together in the same application.

To utilize 64-bits, the Perl executable will need to be recompiled as a 64-bit application. In addition, all libraries, wrapper code, and every other part of your application will need to be compiled for 64-bits. If you plan to use other third-party extension modules, they will also have to be recompiled as 64-bit extensions.

If you are wrapping commercial software for which you have no source code, you will be forced to use the same linking standard as used by that software. This may prevent the use of 64-bit extensions. It may also introduce problems on platforms that support more than one linking standard (e.g., `-o32` and `-n32` on Irix).

## 23.3 Building Perl Extensions under Windows

Building a SWIG extension to Perl under Windows is roughly similar to the process used with Unix. Normally, you will want to produce a DLL that can be loaded into the Perl interpreter. This section assumes you are using SWIG with Microsoft Visual C++ although the procedure may be similar with other compilers.

### 23.3.1 Running SWIG from Developer Studio

If you are developing your application within Microsoft developer studio, SWIG can be invoked as a custom build option. The process roughly requires these steps :

- Open up a new workspace and use the AppWizard to select a DLL project.
- Add both the SWIG interface file (the .i file), any supporting C files, and the name of the wrapper file that will be created by SWIG (ie. `example_wrap.c`). Note : If using C++, choose a different suffix for the wrapper file such as `example_wrap.cxx`. Don't worry if the wrapper file doesn't exist yet—Developer studio will keep a reference to it around.
- Select the SWIG interface file and go to the settings menu. Under settings, select the "Custom Build" option.
- Enter "SWIG" in the description field.
- Enter "`swig -perl5 -o $(ProjDir)\$(InputName)_wrap.cxx $(InputPath)`" in the "Build command(s) field"
- Enter "`$(ProjDir)\$(InputName)_wrap.cxx`" in the "Output files(s) field".
- Next, select the settings for the entire project and go to "C++:Preprocessor". Add the include directories for your Perl 5 installation under "Additional include directories".
- Define the symbols WIN32 and MSWIN32 under preprocessor options. If using the ActiveWare port, also define the symbol PERL\_OBJECT. Note that all extensions to the ActiveWare port must be compiled with the C++ compiler since Perl has been encapsulated in a C++ class.
- Finally, select the settings for the entire project and go to "Link Options". Add the Perl library file to your link libraries. For example "perl.lib". Also, set the name of the output file to match the name of your Perl module (ie. `example.dll`).
- Build your project.

Now, assuming you made it this far, SWIG will be automatically invoked when you build your project. Any changes made to the interface file will result in SWIG being automatically invoked to produce a new version of the wrapper file. To run your new Perl extension, simply run Perl and use the use command as normal. For example :

```
DOS > perl
use example;
$a = example::fact(4);
print "$a\n";
```

### 23.3.2 Using other compilers

SWIG is known to work with Cygwin and may work with other compilers on Windows. For general hints and suggestions refer to the [Windows](#) chapter.

## 23.4 The low-level interface

At its core, the Perl module uses a simple low-level interface to C function, variables, constants, and classes. This low-level interface can be used to control your application. However, it is also used to construct more user-friendly proxy classes as described in the next section.

### 23.4.1 Functions

C functions are converted into new Perl built-in commands (or subroutines). For example:

```
%module example
int fact(int a);
...
```

Now, in Perl:

```
use example;
$a = &example::fact(2);
```

### 23.4.2 Global variables

Global variables are handled using Perl's magic variable mechanism. SWIG generates a pair of functions that intercept read/write operations and attaches them to a Perl variable with the same name as the C global variable. Thus, an interface like this

```
%module example;
...
double Spam;
...
```

is accessed as follows :

```
use example;
print $example::Spam, "\n";
$example::Spam = $example::Spam + 4
# ... etc ...
```

If a variable is declared as `const`, it is wrapped as a read-only variable. Attempts to modify its value will result in an error.

To make ordinary variables read-only, you can also use the `%immutable` directive. For example:

```
%immutable;
extern char *path;
%mutable;
```

The `%immutable` directive stays in effect until it is explicitly disabled using `%mutable`. It is also possible to tag a specific variable as read-only like this:

```
%immutable path;
...
...
extern char *path;          // Declared later in the input
```

### 23.4.3 Constants

Constants are wrapped as read-only Perl variables. For example:

```
%module example

#define FOO 42
```

In Perl:

```
use example;
print $example::FOO, "\n";    # OK
$example::FOO = 2;           # Error
```

## 23.4.4 Pointers

SWIG represents pointers as blessed references. A blessed reference is the same as a Perl reference except that it has additional information attached to it indicating what kind of reference it is. That is, if you have a C declaration like this :

```
Matrix *new_Matrix(int n, int m);
```

The module returns a value generated as follows:

```
$ptr = new_Matrix(int n, int m);      # Save pointer return result
bless $ptr, "p_Matrix";               # Bless it as a pointer to Matrix
```

SWIG uses the "blessing" to check the datatype of various pointers. In the event of a mismatch, an error or warning message is generated.

To check to see if a value is the NULL pointer, use the `defined()` command :

```
if (defined($ptr)) {
    print "Not a NULL pointer.";
} else {
    print "Is a NULL pointer.";
}
```

To create a NULL pointer, you should pass the `undef` value to a function.

The "value" of a Perl reference is not the same as the underlying C pointer that SWIG wrapper functions return. Suppose that `$a` and `$b` are two references that point to the same C object. In general, `$a` and `$b` will be different—since they are different references. Thus, it is a mistake to check the equality of `$a` and `$b` to check the equality of two C pointers. The correct method to check equality of C pointers is to dereference them as follows :

```
if ($$a == $$b) {
    print "a and b point to the same thing in C";
} else {
    print "a and b point to different objects.";
}
```

As much as you might be inclined to modify a pointer value directly from Perl, don't. Manipulating pointer values is architecture dependent and could cause your program to crash. Similarly, don't try to manually cast a pointer to a new type by reblessing a pointer. This may not work like you expect and it is particularly dangerous when casting C++ objects. If you need to cast a pointer or change its value, consider writing some helper functions instead. For example:

```
%inline %{
/* C-style cast */
Bar *FooToBar(Foo *f) {
    return (Bar *) f;
}

/* C++-style cast */
Foo *BarToFoo(Bar *b) {
    return dynamic_cast<Foo*>(b);
}

Foo *IncrFoo(Foo *f, int i) {
    return f+i;
}
%}
```

Also, if working with C++, you should always try to use the new C++ style casts. For example, in the above code, the C-style cast may return a bogus result whereas as the C++-style cast will return NULL if the conversion can't be performed.

**Compatibility Note:** In earlier versions, SWIG tried to preserve the same pointer naming conventions as XS and xsubpp. Given the advancement of the SWIG typesystem and the growing differences between SWIG and XS, this is no longer supported.

## 23.4.5 Structures

Access to the contents of a structure are provided through a set of low-level accessor functions as described in the "SWIG Basics" chapter. For example,

```
struct Vector {
    double x,y,z;
};
```

gets mapped into the following collection of accessor functions:

```
struct Vector *new_Vector();
void          delete_Vector(Vector *v);
double        Vector_x_get(Vector *obj)
void          Vector_x_set(Vector *obj, double x)
double        Vector_y_get(Vector *obj)
void          Vector_y_set(Vector *obj, double y)
double        Vector_z_get(Vector *obj)
void          Vector_z_set(Vector *obj, double z)
```

These functions are then used to access structure data from Perl as follows:

```
$v = example::new_Vector();
print example::Vector_x_get($v), "\n";    # Get x component
example::Vector_x_set($v, 7.8);           # Change x component
```

Similar access is provided for unions and the data members of C++ classes.

const members of a structure are read-only. Data members can also be forced to be read-only using the %immutable directive. For example:

```
struct Foo {
    ...
    %immutable;
    int x;          /* Read-only members */
    char *name;
    %mutable;
    ...
};
```

When char \* members of a structure are wrapped, the contents are assumed to be dynamically allocated using malloc or new (depending on whether or not SWIG is run with the -c++ option). When the structure member is set, the old contents will be released and a new value created. If this is not the behavior you want, you will have to use a typemap (described later).

Array members are normally wrapped as read-only. For example,

```
struct Foo {
    int x[50];
};
```

produces a single accessor function like this:

```
int *Foo_x_get(Foo *self) {
    return self->x;
};
```

If you want to set an array member, you will need to supply a "memberin" typemap described later in this chapter. As a special case, SWIG does generate code to set array members of type `char` (allowing you to store a Python string in the structure).

When structure members are wrapped, they are handled as pointers. For example,

```
struct Foo {
    ...
};

struct Bar {
    Foo f;
};
```

generates accessor functions such as this:

```
Foo *Bar_f_get(Bar *b) {
    return &b->f;
}

void Bar_f_set(Bar *b, Foo *val) {
    b->f = *val;
}
```

### 23.4.6 C++ classes

C++ classes are wrapped by building a set of low level accessor functions. Consider the following class :

```
class List {
public:
    List();
    ~List();
    int  search(char *item);
    void insert(char *item);
    void remove(char *item);
    char *get(int n);
    int  length;
    static void print(List *l);
};
```

When wrapped by SWIG, the following functions are created :

```
List      *new_List();
void      delete_List(List *l);
int       List_search(List *l, char *item);
void      List_insert(List *l, char *item);
void      List_remove(List *l, char *item);
char      *List_get(List *l, int n);
int       List_length_get(List *l);
void      List_length_set(List *l, int n);
void      List_print(List *l);
```

In Perl, these functions are used in a straightforward manner:

```
use example;
$l = example::new_List();
example::List_insert($l,"Ale");
example::List_insert($l,"Stout");
example::List_insert($l,"Lager")
example::List_print($l)
Lager
Stout
Ale
print example::List_length_get($l),"\n";
```



At this low level, C++ objects are really just typed pointers. Member functions are accessed by calling a C-like wrapper with an instance pointer as the first argument. Although this interface is fairly primitive, it provides direct access to C++ objects. A higher level interface using Perl proxy classes can be built using these low-level accessors. This is described shortly.

### 23.4.7 C++ classes and type-checking

The SWIG type-checker is fully aware of C++ inheritance. Therefore, if you have classes like this

```
class Foo {
...
};

class Bar : public Foo {
...
};
```

and a function

```
void spam(Foo *f);
```

then the function `spam( )` accepts `Foo *` or a pointer to any class derived from `Foo`. If necessary, the type-checker also adjusts the value of the pointer (as is necessary when multiple inheritance is used).

### 23.4.8 C++ overloaded functions

If you have a C++ program with overloaded functions or methods, you will need to disambiguate those methods using `%rename`. For example:

```
/* Forward renaming declarations */
%rename(foo_i) foo(int);
%rename(foo_d) foo(double);
...
void foo(int);           // Becomes 'foo_i'
void foo(char *c);       // Stays 'foo' (not renamed)

class Spam {
public:
    void foo(int);        // Becomes 'foo_i'
    void foo(double);     // Becomes 'foo_d'
    ...
};
```

Now, in Perl, the methods are accessed as follows:

```
use example;
example::foo_i(3);
$s = example::new_Spam();
example::Spam_foo_i($s,3);
example::Spam_foo_d($s,3.14);
```

Please refer to the "SWIG Basics" chapter for more information.

### 23.4.9 Operators

C++ operators can also be wrapped using the `%rename` directive. All you need to do is give the operator the name of a valid Perl identifier. For example:

```
%rename(add_complex) operator+(Complex &, Complex &);
...
```

```
Complex operator+(Complex &, Complex &);
```

Now, in Perl, you can do this:

```
use example;
$a = example::new_Complex(2,3);
$b = example::new_Complex(4,-1);
$c = example::add_complex($a,$b);
```

Some preliminary work on mapping C++ operators into Perl operators has been completed. This is covered later.

### 23.4.10 Modules and packages

When you create a SWIG extension, everything gets placed into a single Perl module. The name of the module is determined by the `%module` directive. To use the module, do the following :

```
% perl5
use example;                # load the example module
print example::fact(4),"\n" # Call a function in it
24
```

Usually, a module consists of a collection of code that is contained within a single file. A package, on the other hand, is the Perl equivalent of a namespace. A package is alot like a module, except that it is independent of files. Any number of files may be part of the same package—or a package may be broken up into a collection of modules if you prefer to think about it in this way.

SWIG installs its functions into a package with the same name as the module.

**Incompatible Change:** previous versions of SWIG enabled you to change the name of the package by using the `–package` option, this feature has been removed in order to properly support modules that used nested namespaces, e.g. `Foo::Bar::Baz`. To give your module a nested namespace simply provide the fully qualified name in your `%module` directive:

```
%module "Foo::Bar::Baz"
```

**NOTE:** the double quotes are necessary.

## 23.5 Input and output parameters

A common problem in some C programs is handling parameters passed as simple pointers. For example:

```
void add(int x, int y, int *result) {
    *result = x + y;
}
```

or perhaps

```
int sub(int *x, int *y) {
    return *x+*y;
}
```

The easiest way to handle these situations is to use the `typemaps.i` file. For example:

```
%module example
#include "typemaps.i"

void add(int, int, int *OUTPUT);
int sub(int *INPUT, int *INPUT);
```

In Perl, this allows you to pass simple values. For example:

```
$a = example::add(3,4);
print "$a\n";
7
$b = example::sub(7,4);
print "$b\n";
3
```

Notice how the INPUT parameters allow integer values to be passed instead of pointers and how the OUTPUT parameter creates a return result.

If you don't want to use the names INPUT or OUTPUT, use the %apply directive. For example:

```
%module example
#include "typemaps.i"

%apply int *OUTPUT { int *result };
%apply int *INPUT { int *x, int *y};

void add(int x, int y, int *result);
int sub(int *x, int *y);
```

If a function mutates one of its parameters like this,

```
void negate(int *x) {
    *x = -(*x);
}
```

you can use INOUT like this:

```
%include "typemaps.i"
...
void negate(int *INOUT);
```

In Perl, a mutated parameter shows up as a return value. For example:

```
$a = example::negate(3);
print "$a\n";
-3
```

The most common use of these special typemap rules is to handle functions that return more than one value. For example, sometimes a function returns a result as well as a special error code:

```
/* send message, return number of bytes sent, along with success code */
int send_message(char *text, int len, int *success);
```

To wrap such a function, simply use the OUTPUT rule above. For example:

```
%module example
#include "typemaps.i"
%apply int *OUTPUT { int *success };
...
int send_message(char *text, int *success);
```

When used in Perl, the function will return multiple values.

```
($bytes, $success) = example::send_message("Hello World");
```

Another common use of multiple return values are in query functions. For example:

```
void get_dimensions(Matrix *m, int *rows, int *columns);
```

To wrap this, you might use the following:

```
%module example
#include "typemaps.i"
%apply int *OUTPUT { int *rows, int *columns };
...
void get_dimensions(Matrix *m, int *rows, *columns);
```

Now, in Perl:

```
($r,$c) = example::get_dimensions($m);
```

In certain cases, it is possible to treat Perl references as C pointers. To do this, use the `REFERENCE` typemap. For example:

```
%module example
#include typemaps.i

void add(int x, int y, int *REFERENCE);
```

In Perl:

```
use example;
$c = 0.0;
example::add(3,4,\$c);
print "$c\n";
7
```

**Note:** The `REFERENCE` feature is only currently supported for numeric types (integers and floating point).

## 23.6 Exception handling

The SWIG `%exception` directive can be used to create a user-definable exception handler for converting exceptions in your C/C++ program into Perl exceptions. The chapter on customization features contains more details, but suppose you have a C++ class like the following :

```
class RangeError {}; // Used for an exception

class DoubleArray {
private:
    int n;
    double *ptr;
public:
    // Create a new array of fixed size
    DoubleArray(int size) {
        ptr = new double[size];
        n = size;
    }
    // Destroy an array
    ~DoubleArray() {
        delete ptr;
    }
    // Return the length of the array
    int length() {
        return n;
    }

    // Get an item from the array and perform bounds checking.
    double getitem(int i) {
        if ((i >= 0) && (i < n))
            return ptr[i];
        else
            throw RangeError();
    }
}
```

```
// Set an item in the array and perform bounds checking.
void setitem(int i, double val) {
    if ((i >= 0) && (i < n))
        ptr[i] = val;
    else {
        throw RangeError();
    }
}
};
```

Since several methods in this class can throw an exception for an out-of-bounds access, you might want to catch this in the Perl extension by writing the following in an interface file:

```
%exception {
    try {
        $action
    }
    catch (RangeError) {
        croak("Array index out-of-bounds");
    }
}

class DoubleArray {
    ...
};
```

The exception handling code is inserted directly into generated wrapper functions. The `$action` variable is replaced with the C/C++ code being executed by the wrapper. When an exception handler is defined, errors can be caught and used to gracefully generate a Perl error instead of forcing the entire program to terminate with an uncaught error.

As shown, the exception handling code will be added to every wrapper function. Since this is somewhat inefficient. You might consider refining the exception handler to only apply to specific methods like this:

```
%exception getitem {
    try {
        $action
    }
    catch (RangeError) {
        croak("Array index out-of-bounds");
    }
}

%exception setitem {
    try {
        $action
    }
    catch (RangeError) {
        croak("Array index out-of-bounds");
    }
}
```

In this case, the exception handler is only attached to methods and functions named `getitem` and `setitem`.

If you had a lot of different methods, you can avoid extra typing by using a macro. For example:

```
%define RANGE_ERROR
{
    try {
        $action
    }
    catch (RangeError) {
        croak("Array index out-of-bounds");
    }
}
```

```

}
%enddef

%exception getitem RANGE_ERROR;
%exception setitem RANGE_ERROR;

```

Since SWIG's exception handling is user-definable, you are not limited to C++ exception handling. See the chapter on ["Customization features"](#) for more examples.

**Compatibility note:** In SWIG1.1, exceptions were defined using the older `%except` directive:

```

%except(python) {
    try {
        $function
    }
    catch (RangeError) {
        croak("Array index out-of-bounds");
    }
}

```

This is still supported, but it is deprecated. The newer `%exception` directive provides the same functionality, but it has additional capabilities that make it more powerful.

## 23.7 Remapping datatypes with typemaps

This section describes how you can modify SWIG's default wrapping behavior for various C/C++ datatypes using the `%typemap` directive. This is an advanced topic that assumes familiarity with the Perl C API as well as the material in the ["Typemaps"](#) chapter.

Before proceeding, it should be stressed that typemaps are *not* a required part of using SWIG—the default wrapping behavior is enough in most cases. Typemaps are only used if you want to change some aspect of the primitive C-Perl interface.

### 23.7.1 A simple typemap example

A typemap is nothing more than a code generation rule that is attached to a specific C datatype. For example, to convert integers from Perl to C, you might define a typemap like this:

```

%module example

%typemap(in) int {
    $1 = (int) SvIV($input);
    printf("Received an integer : %d\n", $1);
}
...
extern int fact(int n);

```

Typemaps are always associated with some specific aspect of code generation. In this case, the "in" method refers to the conversion of input arguments to C/C++. The datatype `int` is the datatype to which the typemap will be applied. The supplied C code is used to convert values. In this code a number of special variable prefaced by a `$` are used. The `$1` variable is placeholder for a local variable of type `int`. The `$input` variable is the input object (usually a `SV *`).

When this example is used in Perl5, it will operate as follows :

```

use example;
$n = example::fact(6);
print "$n\n";
...

Output :

```

```
Received an integer : 6
720
```

The application of a typemap to specific datatypes and argument names involves more than simple text-matching—typemaps are fully integrated into the SWIG type-system. When you define a typemap for `int`, that typemap applies to `int` and qualified variations such as `const int`. In addition, the typemap system follows `typedef` declarations. For example:

```
%typemap(in) int n {
    $1 = (int) SvIV($input);
    printf("n = %d\n", $1);
}
typedef int Integer;
extern int fact(Integer n);    // Above typemap is applied
```

It should be noted that the matching of `typedef` only occurs in one direction. If you defined a typemap for `Integer`, it is not applied to arguments of type `int`.

Typemaps can also be defined for groups of consecutive arguments. For example:

```
%typemap(in) (char *str, unsigned len) {
    $1 = SvPV($input, $2);
};

int count(char c, char *str, unsigned len);
```

When a multi-argument typemap is defined, the arguments are always handled as a single Perl object. This allows the function to be used like this (notice how the length parameter is ommitted):

```
example::count("e", "Hello World");
1
>>>
```

### 23.7.2 Perl5 typemaps

The previous section illustrated an "in" typemap for converting Perl objects to C. A variety of different typemap methods are defined by the Perl module. For example, to convert a C integer back into a Perl object, you might define an "out" typemap like this:

```
%typemap(out) int {
    $result = sv_newmortal();
    set_setiv($result, (IV) $1);
    argvi++;
}
```

The following typemap methods are available:

`%typemap(in)`

Converts Perl5 object to input function arguments.

`%typemap(out)`

Converts function return value to a Perl5 value.

`%typemap(varin)`

Converts a Perl5 object to a global variable.

`%typemap(varout)`

Converts a global variable to a Perl5 object.

`%typemap( freearg )`

Cleans up a function argument after a function call

`%typemap( argout )`

Output argument handling

`%typemap( ret )`

Clean up return value from a function.

`%typemap( memberin )`

Setting of C++ member data (all languages).

`%typemap( memberout )`

Return of C++ member data (all languages).

`%typemap( check )`

Check value of input parameter.

### 23.7.3 Typemap variables

Within typemap code, a number of special variables prefaced with a \$ may appear. A full list of variables can be found in the ["Typemaps"](#) chapter. This is a list of the most common variables:

`$1`

A C local variable corresponding to the actual type specified in the `%typemap` directive. For input values, this is a C local variable that's supposed to hold an argument value. For output values, this is the raw result that's supposed to be returned to Perl.

`$input`

A Perl object holding the value of an argument of variable value.

`$result`

A Perl object that holds the result to be returned to Perl.

`$1_name`

The parameter name that was matched.

`$1_type`

The actual C datatype matched by the typemap.

`$1_ltype`



An assignable version of the datatype matched by the typemap (a type that can appear on the left-hand-side of a C assignment operation). This type is stripped of qualifiers and may be an altered version of `$1_type`. All arguments and local variables in wrapper functions are declared using this type so that their values can be properly assigned.

`$symname`

The Perl name of the wrapper function being created.

## 23.7.4 Useful functions

When writing typemaps, it is necessary to work directly with Perl5 objects. This, unfortunately, can be a daunting task. Consult the "perl5" man-page for all of the really ugly details. A short summary of commonly used functions is provided here for reference. It should be stressed that SWIG can be used quite effectively without knowing any of these details—especially now that there are typemap libraries that can already be written.

### Perl Integer Functions

```
int    SvIV(SV *);
void   sv_setiv(SV *sv, IV value);
SV     *newSViv(IV value);
int    SvIOK(SV *);
```

### Perl Floating Point Functions

```
double SvNV(SV *);
void   sv_setnv(SV *, double value);
SV     *newSVnv(double value);
int    SvNOK(SV *);
```

### Perl String Functions

```
char    *SvPV(SV *, STRLEN len);
void    sv_setpv(SV *, char *val);
void    sv_setpvn(SV *, char *val, STRLEN len);
SV      *newSVpv(char *value, STRLEN len);
int     SvPOK(SV *);
void    sv_catpv(SV *, char *);
void    sv_catpvn(SV *, char *, STRLEN);
```

### Perl References

```
void    sv_setref_pv(SV *, char *, void *ptr);
int     sv_isobject(SV *);
SV      *SvRV(SV *);
int     sv_isa(SV *, char *0);
```

## 23.8 Typemap Examples

This section includes a few examples of typemaps. For more examples, you might look at the files "perl5.swg" and "typemaps.i" in the SWIG library.

### 23.8.1 Converting a Perl5 array to a char \*\*

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Perl5 array reference to be used as a `char **` datatype.

```
%module argv
```

## SWIG-1.3 Documentation

```
// This tells SWIG to treat char ** as a special case
%typemap(in) char ** {
    AV *tempav;
    I32 len;
    int i;
    SV **tv;
    if (!SvROK($input))
        croak("Argument $argnum is not a reference.");
    if (SvTYPE(SvRV($input)) != SVt_PVAV)
        croak("Argument $argnum is not an array.");
    tempav = (AV*)SvRV($input);
    len = av_len(tempav);
    $1 = (char **) malloc((len+2)*sizeof(char *));
    for (i = 0; i <= len; i++) {
        tv = av_fetch(tempav, i, 0);
        $1[i] = (char *) SvPV(*tv, PL_na);
    }
    $1[i] = NULL;
};

// This cleans up the char ** array after the function call
%typemap(freearg) char ** {
    free($1);
}

// Creates a new Perl array and places a NULL-terminated char ** into it
%typemap(out) char ** {
    AV *myav;
    SV **svs;
    int i = 0, len = 0;
    /* Figure out how many elements we have */
    while ($1[len])
        len++;
    svs = (SV **) malloc(len*sizeof(SV *));
    for (i = 0; i < len ; i++) {
        svs[i] = sv_newmortal();
        sv_setpv((SV*)svs[i], $1[i]);
    };
    myav = av_make(len, svs);
    free(svs);
    $result = newRV((SV*)myav);
    sv_2mortal($result);
    argvi++;
}

// Now a few test functions
%inline %{
int print_args(char **argv) {
    int i = 0;
    while (argv[i]) {
        printf("argv[%d] = %s\n", i, argv[i]);
        i++;
    }
    return i;
}

// Returns a char ** list
char **get_args() {
    static char *values[] = { "Dave", "Mike", "Susan", "John", "Michelle", 0 };
    return &values[0];
}
%}
```

When this module is compiled, the wrapped C functions can be used in a Perl script as follows :

```
use argv;
```

```

@a = ("Dave", "Mike", "John", "Mary");
argv::print_args(\@a);
$b = argv::get_args();
print @$b, "\n";

# Create an array of strings
# Pass it to our C function
# Get array of strings from C
# Print it out

```

### 23.8.2 Return values

Return values are placed on the argument stack of each wrapper function. The current value of the argument stack pointer is contained in a variable `argvi`. Whenever a new output value is added, it is critical that this value be incremented. For multiple output values, the final value of `argvi` should be the total number of output values.

The total number of return values should not exceed the number of input values unless you explicitly extend the argument stack. This can be done using the `EXTEND()` macro as in :

```

%typemap(argout) int *OUTPUT {
    if (argvi >= items) {
        EXTEND(sp,1);          /* Extend the stack by 1 object */
    }
    $result = sv_newmortal();
    sv_setiv($target, (IV) *($1));
    argvi++;
}

```

### 23.8.3 Returning values from arguments

Sometimes it is desirable for a function to return a value in one of its arguments. This example describes the implementation of the `OUTPUT` typemap.

```

%module return

// This tells SWIG to treat an double * argument with name 'OutDouble' as
// an output value.

%typemap(argout) double *OUTPUT {
    $result = sv_newmortal();
    sv_setnv($result, *$input);
    argvi++;          /* Increment return count -- important! */
}

// We don't care what the input value is. Ignore, but set to a temporary variable

%typemap(in,numinputs=0) double *OUTPUT(double junk) {
    $1 = &junk;
}

// Now a function to test it
%{
/* Returns the first two input arguments */
int multout(double a, double b, double *out1, double *out2) {
    *out1 = a;
    *out2 = b;
    return 0;
};
%}

// If we name both parameters OutDouble both will be output

int multout(double a, double b, double *OUTPUT, double *OUTPUT);
...

```

When this function is called, the output arguments are appended to the stack used to return results. This shows up an array in Perl. For example :

```
@r = multout(7,13);
print "multout(7,13) = @r\n";
($x,$y) = multout(7,13);
```

### 23.8.4 Accessing array structure members

Consider the following data structure :

```
#define SIZE 8
typedef struct {
    int    values[SIZE];
    ...
} Foo;
```

By default, SWIG doesn't know how to handle the values structure member it's an array, not a pointer. In this case, SWIG makes the array member read-only. Reading will simply return a pointer to the first item in the array. To make the member writable, a "memberin" typemap can be used.

```
%typemap(memberin) int [SIZE] {
    int i;
    for (i = 0; i < SIZE; i++) {
        $1[i] = $input[i];
    }
}
```

Whenever a `int [SIZE]` member is encountered in a structure or class, this typemap provides a safe mechanism for setting its value.

As in the previous example, the typemap can be generalized for any dimension. For example:

```
%typemap(memberin) int [ANY] {
    int i;
    for (i = 0; i < $1_dim0; i++) {
        $1[i] = $input[i];
    }
}
```

When setting structure members, the input object is always assumed to be a C array of values that have already been converted from the target language. Because of this, the `memberin` typemap is almost always combined with the use of an "in" typemap. For example, the "in" typemap in the previous section would be used to convert an `int [ ]` array to C whereas the "memberin" typemap would be used to copy the converted array into a C data structure.

### 23.8.5 Turning Perl references into C pointers

A frequent confusion on the SWIG mailing list is errors caused by the mixing of Perl references and C pointers. For example, suppose you have a C function that modifies its arguments like this :

```
void add(double a, double b, double *c) {
    *c = a + b;
}
```

A common misinterpretation of this function is the following Perl script :

```
# Perl script
$a = 3.5;
$b = 7.5;
$c = 0.0;          # Output value
add($a,$b,\$c);    # Place result in c (Except that it doesn't work)
```

To make this work with a reference, you can use a typemap such as this:

```
%typemap(in) double * (double dvalue) {
    SV* tempSV;
    if (!SvROK($input)) {
        croak("expected a reference\n");
    }
    tempSV = SvRV($input);
    if ((!SvNOK(tempSV)) && (!SvIOK(tempSV))) {
        croak("expected a double reference\n");
    }
    dvalue = SvNV(tempSV);
    $1 = &dvalue;
}

%typemap(argout) double * {
    SV *tempSV;
    tempSV = SvRV($input);
    sv_setnv(tempSV, *$input);
}
```

Now, if you place this before the add function, you can do this :

```
$a = 3.5;
$b = 7.5;
$c = 0.0;
add($a,$b,$c);          # Now it works!
print "$c\n";
```

## 23.8.6 Pointer handling

Occasionally, it might be necessary to convert pointer values that have been stored using the SWIG typed-pointer representation. To convert a pointer from Perl to C, the following function is used:

```
int SWIG_ConvertPtr(SV *obj, void **ptr, swig_type_info *ty, int flags)
```

Converts a Perl object `obj` to a C pointer. The result of the conversion is placed into the pointer located at `ptr`. `ty` is a SWIG type descriptor structure. `flags` is used to handle error checking and other aspects of conversion. `flags` is currently undefined and reserved for future expansion. Returns 0 on success and -1 on error.

```
void *SWIG_MakePtr(SV *obj, void *ptr, swig_type_info *ty, int flags)
```

Creates a new Perl pointer object. `obj` is a Perl SV that has been initialized to hold the result, `ptr` is the pointer to convert, `ty` is the SWIG type descriptor structure that describes the type, and `flags` is a flag that controls properties of the conversion. `flags` is currently undefined and reserved.

Both of these functions require the use of a special SWIG type-descriptor structure. This structure contains information about the mangled name of the datatype, type-equivalence information, as well as information about converting pointer values under C++ inheritance. For a type of `Foo *`, the type descriptor structure is usually accessed as follows:

```
Foo *f;
if (SWIG_ConvertPtr($input, (void **) &f, SWIGTYPE_p_Foo, 0) == -1) return NULL;

SV *sv = sv_newmortal();
SWIG_MakePtr(sv, f, SWIGTYPE_p_Foo, 0);
```

In a typemap, the type descriptor should always be accessed using the special typemap variable `$1_descriptor`. For example:

```
%typemap(in) Foo * {
```

```

    if ((SWIG_ConvertPtr($input,(void **) &$1, $1_descriptor,0)) == -1) return NULL;
}

```

If necessary, the descriptor for any type can be obtained using the `$descriptor()` macro in a typemap. For example:

```

%typemap(in) Foo * {
    if ((SWIG_ConvertPtr($input,(void **) &$1, $descriptor(Foo *), 0)) == -1) return NULL;
}

```

## 23.9 Proxy classes

**Out of date. Needs update.**

Using the low-level procedural interface, SWIG can also construct a high-level object oriented interface to C structures and C++ classes. This is done by constructing a Perl proxy class (also known as a shadow class) that provides an OO wrapper to the underlying code. This section describes the implementation details of the proxy interface.

### 23.9.1 Preliminaries

Proxy classes, are generated by default. If you want to turn them off, use the `-noproxy` command line option. For example:

```
$ swig -c++ -perl -noproxy example.i
```

When proxy classes are used, SWIG moves all of the low-level procedural wrappers to another package name. By default, this package is named 'modulec' where 'module' is the name of the module you provided with the `%module` directive. Then, in place of the original module, SWIG creates a collection of high-level Perl wrappers. In your scripts, you will use these high level wrappers. The wrappers, in turn, interact with the low-level procedural module.

### 23.9.2 Structure and class wrappers

Suppose you have the following SWIG interface file :

```

%module example
struct Vector {
    Vector(double x, double y, double z);
    ~Vector();
    double x,y,z;
};

```

When wrapped, SWIG creates the following set of low-level accessor functions as described in previous sections.

```

Vector *new_Vector(double x, double y, double z);
void    delete_Vector(Vector *v);
double  Vector_x_get(Vector *v);
double  Vector_x_set(Vector *v, double value);
double  Vector_y_get(Vector *v);
double  Vector_y_set(Vector *v, double value);
double  Vector_z_get(Vector *v);
double  Vector_z_set(Vector *v, double value);

```

However, when proxy classes are enabled, these accessor functions are wrapped inside a Perl class like this:

```

package example::Vector;
@ISA = qw( example );
%OWNER = ();
%BLESSEDMEMBERS = ();

sub new () {

```

```

my $self = shift;
my @args = @_;
$self = vectorc::new_Vector(@args);
return undef if (!defined($self));
bless $self, "example::Vector";
$OWNER{$self} = 1;
my %retval;
tie %retval, "example::Vector", $self;
return bless \%retval, "Vector";
}

sub DESTROY {
    return unless $_[0]->isa('HASH');
    my $self = tied(%{$_[0]});
    delete $ITERATORS{$self};
    if (exists $OWNER{$self}) {
        examplec::delete_Vector($self);
        delete $OWNER{$self};
    }
}

sub FETCH {
    my ($self,$field) = @_;
    my $member_func = "vectorc::Vector_{$field}_get";
    my $val = &$member_func($self);
    if (exists $BLESSEDMEMBERS{$field}) {
        return undef if (!defined($val));
        my %retval;
        tie %retval,$BLESSEDMEMBERS{$field},$val;
        return bless \%retval, $BLESSEDMEMBERS{$field};
    }
    return $val;
}

sub STORE {
    my ($self,$field,$newval) = @_;
    my $member_func = "vectorc::Vector_{$field}_set";
    if (exists $BLESSEDMEMBERS{$field}) {
        &$member_func($self,tied(%{$newval}));
    } else {
        &$member_func($self,$newval);
    }
}

```

Each structure or class is mapped into a Perl package of the same name. The C++ constructors and destructors are mapped into constructors and destructors for the package and are always named "new" and "DESTROY". The constructor always returns a tied hash table. This hash table is used to access the member variables of a structure in addition to being able to invoke member functions. The %OWNER and %BLESSEDMEMBERS hash tables are used internally and described shortly.

To use our new proxy class we can simply do the following:

```

# Perl code using Vector class
$v = new Vector(2,3,4);
$w = Vector->new(-1,-2,-3);

# Assignment of a single member
$v->{x} = 7.5;

# Assignment of all members
%$v = ( x=>3,
        y=>9,
        z=>-2);

# Reading members
$x = $v->{x};

# Destruction

```

```
$v->DESTROY();
```

### 23.9.3 Object Ownership

In order for proxy classes to work properly, it is necessary for Perl to manage some mechanism of object ownership. Here's the crux of the problem—suppose you had a function like this :

```
Vector *Vector_get(Vector *v, int index) {
    return &v[i];
}
```

This function takes a Vector pointer and returns a pointer to another Vector. Such a function might be used to manage arrays or lists of vectors (in C). Now contrast this function with the constructor for a Vector object :

```
Vector *new_Vector(double x, double y, double z) {
    Vector *v;
    v = new Vector(x,y,z);      // Call C++ constructor
    return v;
}
```

Both functions return a Vector, but the constructor is returning a brand-new Vector while the other function is returning a Vector that was already created (hopefully). In Perl, both vectors will be indistinguishable—clearly a problem considering that we would probably like the newly created Vector to be destroyed when we are done with it.

To manage these problems, each class contains two methods that access an internal hash table called %OWNER. This hash keeps a list of all of the objects that Perl knows that it has created. This happens in two cases: (1) when the constructor has been called, and (2) when a function implicitly creates a new object (as is done when SWIG needs to return a complex datatype by value). When the destructor is invoked, the Perl proxy class module checks the %OWNER hash to see if Perl created the object. If so, the C/C++ destructor is invoked. If not, we simply destroy the Perl object and leave the underlying C object alone (under the assumption that someone else must have created it).

This scheme works remarkably well in practice but it isn't foolproof. In fact, it will fail if you create a new C object in Perl, pass it on to a C function that remembers the object, and then destroy the corresponding Perl object (this situation turns out to come up frequently when constructing objects like linked lists and trees). When C takes possession of an object, you can change Perl's ownership by simply deleting the object from the %OWNER hash. This is done using the DISOWN method.

```
# Perl code to change ownership of an object
$v = new Vector(x,y,z);
$v->DISOWN();
```

To acquire ownership of an object, the ACQUIRE method can be used.

```
# Given Perl ownership of a file
$u = Vector_get($v);
$u->ACQUIRE();
```

As always, a little care is in order. SWIG does not provide reference counting, garbage collection, or advanced features one might find in sophisticated languages.

### 23.9.4 Nested Objects

Suppose that we have a new object that looks like this :

```
struct Particle {
    Vector r;
    Vector v;
    Vector f;
    int    type;
```



```
}
```

In this case, the members of the structure are complex objects that have already been encapsulated in a Perl proxy class. To handle these correctly, we use the `%BLESSEDMEMBERS` hash which would look like this (along with some supporting code) :

```
package Particle;
...
%BLESSEDMEMBERS = (
    r => `Vector`,
    v => `Vector`,
    f => `Vector`,
);
```

When fetching members from the structure, `%BLESSEDMEMBERS` is checked. If the requested field is present, we create a tied-hash table and return it. If not, we just return the corresponding member unmodified.

This implementation allows us to operate on nested structures as follows :

```
# Perl access of nested structure
$p = new Particle();
$p->{f}->{x} = 0.0;
%{$p->{v}} = ( x=>0, y=>0, z=>0);
```

### 23.9.5 Proxy Functions

When functions take arguments involving a complex object, it is sometimes necessary to write a proxy function. For example :

```
double dot_product(Vector *v1, Vector *v2);
```

Since `Vector` is an object already wrapped into a proxy class, we need to modify this function to accept arguments that are given in the form of tied hash tables. This is done by creating a Perl function like this :

```
sub dot_product {
    my @args = @_;
    $args[0] = tied(%{$args[0]});      # Get the real pointer values
    $args[1] = tied(%{$args[1]});
    my $result = vectorc::dot_product(@args);
    return $result;
}
```

This function replaces the original function, but operates in an identical manner.

### 23.9.6 Inheritance

Simple C++ inheritance is handled using the Perl `@ISA` array in each class package. For example, if you have the following interface file :

```
// shapes.i
// SWIG interface file for shapes class
%module shapes
%{
#include "shapes.h"
%}

class Shape {
public:
    virtual double area() = 0;
    virtual double perimeter() = 0;
    void    set_location(double x, double y);
};
```

```

class Circle : public Shape {
public:
    Circle(double radius);
    ~Circle();
    double area();
    double perimeter();
};
class Square : public Shape {
public:
    Square(double size);
    ~Square();
    double area();
    double perimeter();
}

```

The resulting, Perl wrapper class will create the following code :

```

Package Shape;
@ISA = (shapes);
...
Package Circle;
@ISA = (shapes Shape);
...
Package Square;
@ISA = (shapes Shape);

```

The @ISA array determines where to look for methods of a particular class. In this case, both the Circle and Square classes inherit functions from Shape so we'll want to look in the Shape base class for them. All classes also inherit from the top-level module shapes. This is because certain common operations needed to implement proxy classes are implemented only once and reused in the wrapper code for various classes and structures.

Since SWIG proxy classes are implemented in Perl, it is easy to subclass from any SWIG generated class. To do this, simply put the name of a SWIG class in the @ISA array for your new class. However, be forewarned that this is not a trivial problem. In particular, inheritance of data members is extremely tricky (and I'm not even sure if it really works).

### 23.9.7 Modifying the proxy methods

It is possible to override the SWIG generated proxy/shadow methods, using `%feature("shadow")`. It works like all the other [%feature directives](#). Here is a simple example showing how to add some Perl debug code to the constructor:

```

/* Let's make the constructor of the class Square more verbose */
%feature("shadow") Square(double w)
%{
    sub new {
        my $pkg = shift;
        my $self = examplec::new_Square(@_);
        print STDERR "Constructed an @ {[ref($self)]}\n";
        bless $self, $pkg if defined($self);
    }
}%

class Square {
public:
    Square(double w);
    ...
};

```

## 24 SWIG and PHP4

- [Preliminaries](#)
- [Building PHP4 Extensions](#)
  - ◆ [Building a loadable extension](#)
  - ◆ [Basic PHP4 interface](#)
  - ◆ [Functions](#)
  - ◆ [Global Variables](#)
  - ◆ [Pointers](#)
  - ◆ [Structures and C++ classes](#)
  - ◆ [Constants](#)
  - ◆ [Proxy classes](#)
  - ◆ [Constructors and Destructors](#)
  - ◆ [Static Member Variables](#)
  - ◆ [PHP4 Pragmas](#)
  - ◆ [Building extensions into php](#)
  - ◆ [To be furthered...](#)

**Caution:** This chapter (and module!) is still under construction

In this chapter, we discuss SWIG's support of PHP4. The PHP4 module is still under development so some of the features below may not work properly (or at all!.)

The PHP4 module has undergone a lot of changes recently affecting the way proxy classes are implemented so you should read this document even if you thought you were familiar with what it said. The major change is that proxy classes are implemented inside the php module in C++ instead of in the generated .php file in php.

### 24.1 Preliminaries

In order to use this module, you will need to have a copy of the PHP 4.0 (or above) include files to compile the SWIG generated files. You can find these files by running 'php-config --includes'. To test the modules you will need either the php binary or the Apache php module. If you want to build your extension into php directly (without having the overhead of loading it into each script), you will need the complete PHP source tree available.

### 24.2 Building PHP4 Extensions

To build a PHP4 extension, run swig using the `-php4` option as follows :

```
swig -php4 example.i
```

This will produce 3 files by default. The first file, `example_wrap.c` contains all of the C code needed to build a PHP4 extension. The second file, `php_example.h` contains the header information needed to link the extension into PHP. The third file, `example.php` can be included by your php scripts. It will attempt to dynamically load your extension, and is a place-holder for extra code specified in the interface file. If you want to build your extension using the `phpize` utility, or if you want to build your module into PHP directly, you can specify the `-phpfull` command line argument to swig.

The `-phpfull` will generate three extra files. The first extra file, `config.m4` contains the shell code needed to enable the extension as part of the PHP4 build process. The second extra file, `Makefile.in` contains the information needed to build the final Makefile after substitutions. The third and final extra file, `CREDITS` should contain the credits for the extension.

To finish building the extension, you have two choices. You can either build the extension as a separate object file which will then have to be explicitly loaded by each script. Or you can rebuild the entire php source tree and build the extension into the php executable/library so it will be available in every script. The first choice is the default, however it can be changed by passing the `'-phpfull'` command line switch to select the second build method.

## 24.2.1 Building a loadable extension

To build a dynamic module for PHP, you have two options. You can use the `phpize` utility, or you can do it manually.

To build manually, use a compile string similar to this (different for each OS):

```
cc -I.. $(PHPINC) -fpic -c example_wrap.c
cc -shared example_wrap.o -o libexample.so
```

To build with `phpize`, after you have run `swig` you will need to run the 'phpize' command (installed as part of php) in the same directory. This re-creates the php build environment in that directory. It also creates a configure file which includes the shell code from the `config.m4` that was generated by SWIG, this configure script will accept a command line argument to enable the extension to be run ( by default the command line argument is `--enable-modulename`, however you can edit the `config.m4` file before running `phpize` to accept `--with-modulename`. You can also add extra tests in `config.m4` to check that a correct library version is installed or correct header files are included, etc, but you must edit this file before running `phpize`. ) If you like SWIG can generate simple extra tests for libraries and header files for you.

```
swig -php4 -phpfull -withlibs "xapian omquery" --withincs "om.h"
```

Will include in the `config.m4` search for `libxapian.a` or `libxapian.so` and search for `libomquery.a` or `libomquery.so` as well as a search for `om.h`

If you depend on source files not generated by SWIG, before generated configure file, you may need to edit the `Makefile.in` file. This contains the names of the source files to compile (just the wrapper file by default) and any additional libraries needed to be linked in. If there are extra C files to compile, you will need to add them to the `Makefile.in`, or add the names of libraries if they are needed. In simple cases SWIG is pretty good at generating a complete `Makefile.in` and `config.m4` which need no further editing.

You then run the configure script with the command line argument needed to enable the extension. Then run `make`, which builds the extension. The extension object file will be left in the modules sub directory, you can move it to wherever it is convenient to call from your php script.

To test the extension from a PHP script, you need to load it first. You do this by putting the line,

```
dl("/path/to/modulename.so"); // Load the module
```

at the start of each PHP file. SWIG also generates a php module, which attempts to do the `dl ( )` call for you:

```
include("example.php");
```

A more complicated method which builds the module directly into the php executable is described [below](#).

## 24.2.2 Basic PHP4 interface

## 24.2.3 Functions

C functions are converted into PHP functions. Default/optional arguments are also allowed. An interface file like this :

```
%module default
int foo(int a);
double bar(double, double b = 3.0);
...
```

Will be accessed in PHP like this :

```
dl("default.so"); $a = foo(2);
$b = bar(3.5, -1.5);
$c = bar(3.5);      # Use default argument for 2nd parameter
```

## 24.2.4 Global Variables

Global variables are difficult for PHP to handle, unlike Perl, there is no 'magic' way to intercept modifications made to variables, so changes in a PHP variable will not be reflected in its C equivalent. To get around the problem, two extra functions are generated, `Swig_sync_c()` and `Swig_sync_php()`. These functions are called at the start and end of every function call, ensuring changes made in PHP are updated in C (and vice versa.) Because this is handled for you, you can modify the variables in PHP as normal, e.g.

```
%module example;
...
double seki = 2;
...
int example_func(void);
```

is accessed as follow :

```
dl("example.so");
print $seki;
$seki = $seki * 2;      # Does not affect C variable, still equal to 2
example_func();        # Syncs C variable to PHP Variable, now both 4
```

SWIG supports global variables of all C datatypes including pointers and complex objects.

## 24.2.5 Pointers

Pointers to C/C++ objects **no longer** represented as character strings such as `_523d3f4_Circle_p`, instead they are represented as PHP resources, rather like MySQL connection handles.

You can also explicitly create a NULL pointer as a string "NULL" or by passing a null or empty value.

## 24.2.6 Structures and C++ classes

For structures and classes, SWIG produces accessor functions for each member function and data. For example :

```
%module vector

class Vector {
public:
    double x,y,z;
    Vector();
    ~Vector();
    double magnitude();
};
```

This gets turned into the following collection of PHP functions :

```
Vector_x_set($obj);
Vector_x_get($obj);
Vector_y_set($obj);
Vector_y_get($obj);
Vector_z_set($obj);
Vector_z_get($obj);
new_Vector();
delete_Vector($obj);
Vector_magnitude($obj);
```

To use the class, simply use these functions. However, SWIG also has a mechanism for creating proxy classes that hides these functions and uses an object oriented interface instead – see [below](#)

## 24.2.7 Constants

These work in much the same way as in C/C++, constants can be defined by using either the normal C pre-processor declarations, or the `%constant` SWIG directive. These will then be available from your PHP script as a PHP constant, (e.g. no dollar sign is needed to access them. ) For example, with a swig file like this,

```
%module example

#define PI 3.14159

%constant int E = 2.71828
```

you can access from in your php script like this,

```
dl("libexample.so");

echo "PI = " . PI . "\n";

echo "E = " . E . "\n";
```

There are two peculiarities with using constants in PHP4. The first is that if you try to use an undeclared constant, it will evaluate to a string set to the constants name. For example,

```
%module example

#define EASY_TO_MISPELL 0
```

accessed incorrectly in PHP,

```
dl("libexample.so");

if(EASY_TO_MISPEL) {
    ....
} else {
    ....
}
```

will issue a warning about the undeclared constant, but will then evaluate it and turn it into a string ('EASY\_TO\_MISPEL'), which evaluates to true, rather than the value of the constant which would be false. This is a feature.

The second 'feature' is that although constants are case sensitive (by default), you cannot declare a constant twice with alternative cases. E.g.,

```
%module example

#define TEST    Hello
#define Test    World
```

accessed from PHP,

```
dl("libexample.so");

echo TEST, Test;
```

will output "Hello Test" rather than "Hello World". This is because internally, all constants are stored in a hash table by their lower case name, so 'TEST' and 'Test' will map to the same hash element ('Test'). But, because we declared them case sensitive,

the Zend engine will test if the case matches with the case the constant was declared with first.

So, in the example above, the TEST constant was declared first, and will be stored under the hash element 'test'. The 'Test' constant will also map to the same hash element 'test', but will not overwrite it. When called from the script, the TEST constant will again be mapped to the hash element 'test' so the constant will be retrieved. The case will then be checked, and will match up, so the value ('Hello') will be returned. When 'Test' is evaluated, it will also map to the same hash element 'test'. The same constant will be retrieved, this time though the case check will fail as 'Test' != 'TEST'. So PHP will assume that Test is a undeclared constant, and as explained above, will return it as a string set to the constant name ('Test'). Hence the script above will print 'Hello Test'. If they were declared non-case sensitive, the output would be 'Hello Hello', as both point to the same value, without the case test taking place. ( Apologies, this paragraph needs rewriting to make some sense. )

## 24.2.8 Proxy classes

To avoid having to call the various accessor function to get at structures or class members, we can turn C structs and C++ classes into PHP classes that can be used directly in PHP scripts as objects and object methods. This is done by writing additional PHP code that builds PHP classes on top of the low-level SWIG interface. These PHP classes shadow or proxy an underlying C/C++ class.

To have SWIG create proxy classes, use the `-proxy` option :

```
% swig -php4 -proxy tbc.i
```

This will produce the same files as before except that the final module will declare internal PHP classes with the same names as the classes in your .i file. No longer are the proxy classes defined in the .php file, it will not contain significantly more support PHP code.

For the most part, the code is the same except that we can now access members of complex data structures using `->` instead of the low level access or functions like before.

.... ( more examples on the way ) ....

## 24.2.9 Constructors and Destructors

Constructors are used in PHP as in C++, they are called when the object is created and any arguments are passed to them. However, function overloading is not allowed in PHP so only one constructor can be used. This creates a problem when copying objects, as we cannot avoid creating a whole new one when all we want is to make it point to the same value as the original. This is currently worked around by doing the following,

- Create the new PHP object
- Delete the PHP objects pointer to the C object
- Set the PHP object's pointer to the same as the original PHP object's pointer.

This is rather convoluted and hopefully will be improved upon in a later release.

Because the internal wrapped objects are wrapped in PHP resources, PHP handles the cleaning up when there are no more references to the wrapped object. 'RegisterShutdownFunction' is no longer needed for this. I am not sure if PHP resources are all freed at the end of a script, or when they each go out of scope.

## 24.2.10 Static Member Variables

Class variables are not supported in PHP, however class functions are, using '::' syntax. Static member variables are therefore accessed using a class function with the same name, which returns the current value of the class variable. For example

```
%module example

class Ko {
    static int threats;
```

```
...
};
```

would be accessed in PHP as,

```
dl("libexample.so");

echo "There has now been " . Ko::threats() . " threats\n";
```

To set the static member variable, pass the value as the argument to the class function, e.g.

```
Ko::threats(10);

echo "There has now been " . Ko::threats() . " threats\n";
```

### 24.2.11 PHP4 Pragmas

There are a few pragmas understood by the PHP4 module. The first, **include** adds a file to be included by the generated PHP module. The second, **code** adds literal code to the generated PHP module. The third, **phpinfo** inserts code to the function called when PHP's `phpinfo()` function is called.

```
/* example.i */

#pragma/php4 include="foo.php"
#pragma/php4 code="
    function foo($bar) {
        /* do something */
    }
"
#pragma/php4 phpinfo="
    zend_printf("An example of PHP support through SWIG\n");
    php_info_print_table_start();
    php_info_print_table_header(2, "Directive", "Value");
    php_info_print_table_row(2, "Example support", "enabled");
    php_info_print_table_end();
"

#include "example.h"
```

### 24.2.12 Building extensions into php

This method, selected with the `-phpfull` command line switch, involves rebuilding the entire php source tree. Whilst more complicated to build, it does mean that the extension is then available without having to load it in each script.

After running `swig` with the `-phpfull` switch, you will be left with a shockingly similiar set of files to the previous build process. However you will then need to move these files to a subdirectory within the php source tree, this subdirectory you will need to create under the `ext` directory, with the name of the extension ( e.g `mkdir php-4.0.6/ext/modulename` )

After moving the files into this directory, you will need to run the 'buildall' script in the php source directory. This rebuilds the configure script and includes the extra command line arguments from the module you have added.

Before running the generated configure file, you may need to edit the `Makefile.in`. This contains the names of the source files to compile ( just the wrapper file by default) and any additional libraries needed to link in. If their are extra C files to compile you will need to add them to the `Makefile`, or add the names of libraries if they are needed. In most cases `Makefile.in` will be complete, especially if you make use of `-withlibs` and `-withincs`

```
swig -php4 -phpfull -withlibs "xapian omquery" --withincs "om.h"
```



## SWIG-1.3 Documentation

Will include in the config.m4 and Makefile.in search for libxapian.a or libxapian.so and search for libomquery.a or libomquery.so as well as a search for om.h

You then need to run the configure command and pass the necessary command line arguments to enable your module ( by default this is `--enable-modulename`, but this can be changed by editing the config.m4 file in the modules directory before running the buildall script. In addition, extra tests can be added to the config.m4 file to ensure the correct libraries and header files are installed.)

Once configure has completed, you can run make to build php. If this all compiles correctly, you should end up with a php executable/library which contains your new module. You can test it with a php script which does not have the 'dl' command as used above.

### **24.2.13 To be furthered...**

## 25 SWIG and Pike

- [Preliminaries](#)
  - ◆ [Running SWIG](#)
  - ◆ [Getting the right header files](#)
  - ◆ [Using your module](#)
- [Basic C/C++ Mapping](#)
  - ◆ [Modules](#)
  - ◆ [Functions](#)
  - ◆ [Global variables](#)
  - ◆ [Constants and enumerated types](#)
  - ◆ [Constructors and Destructors](#)
  - ◆ [Static Members](#)

This chapter describes SWIG support for Pike. As of this writing, the SWIG Pike module is still under development and is not considered ready for prime time. The Pike module is being developed against the Pike 7.4.10 release and may not be compatible with previous versions of Pike.

This chapter covers most SWIG features, but certain low-level details are covered in less depth than in earlier chapters. At the very least, make sure you read the "[SWIG Basics](#)" chapter.

### 25.1 Preliminaries

#### 25.1.1 Running SWIG

Suppose that you defined a SWIG module such as the following:

```
%module example

%{
#include "example.h"
%}

int fact(int n);
```

To build a C extension module for Pike, run SWIG using the `-pike` option :

```
$ swig -pike example.i
```

If you're building a C++ extension, be sure to add the `-c++` option:

```
$ swig -c++ -pike example.i
```

This creates a single source file named `example_wrap.c` (or `example_wrap.cxx`, if you ran SWIG with the `-c++` option). The SWIG-generated source file contains the low-level wrappers that need to be compiled and linked with the rest of your C/C++ application to create an extension module.

The name of the wrapper file is derived from the name of the input file. For example, if the input file is `example.i`, the name of the wrapper file is `example_wrap.c`. To change this, you can use the `-o` option:

```
$ swig -pike -o pseudonym.c example.i
```

#### 25.1.2 Getting the right header files

In order to compile the C/C++ wrappers, the compiler needs to know the path to the Pike header files. These files are usually contained in a directory such as

```
/usr/local/pike/7.4.10/include/pike
```

There doesn't seem to be any way to get Pike itself to reveal the location of these files, so you may need to hunt around for them. You're looking for files with the names `global.h`, `program.h` and so on.

### 25.1.3 Using your module

To use your module, simply use Pike's `import` statement:

```
$ pike
Pike v7.4 release 10 running Hilfe v3.5 (Incremental Pike Frontend)
> import example;
> fact(4);
(1) Result: 24
```

## 25.2 Basic C/C++ Mapping

### 25.2.1 Modules

All of the code for a given SWIG module is wrapped into a single Pike module. Since the name of the shared library that implements your module ultimately determines the module's name (as far as Pike is concerned), SWIG's `%module` directive doesn't really have any significance.

### 25.2.2 Functions

Global functions are wrapped as new Pike built-in functions. For example,

```
%module example

int fact(int n);
```

creates a new built-in function `example.fact(n)` that works exactly as you'd expect it to:

```
> import example;
> fact(4);
(1) Result: 24
```

### 25.2.3 Global variables

Global variables are currently wrapped as a pair of functions, one to get the current value of the variable and another to set it. For example, the declaration

```
%module example

double Foo;
```

will result in two functions, `Foo_get()` and `Foo_set()`:

```
> import example;
> Foo_get();
(1) Result: 3.000000
> Foo_set(3.14159);
(2) Result: 0
> Foo_get();
(3) Result: 3.141590
```

## 25.2.4 Constants and enumerated types

Enumerated types in C/C++ declarations are wrapped as Pike constants, not as Pike enums.

## 25.2.5 Constructors and Destructors

Constructors are wrapped as `create()` methods, and destructors are wrapped as `destroy()` methods, for Pike classes.

## 25.2.6 Static Members

Since Pike doesn't support static methods or data for Pike classes, static member functions in your C++ classes are wrapped as regular functions and static member variables are wrapped as pairs of functions (one to get the value of the static member variable, and another to set it). The names of these functions are prepended with the name of the class. For example, given this C++ class declaration:

```
class Shape
{
public:
    static void print();
    static int nshapes;
};
```

SWIG will generate a `Shape_print()` method that invokes the static `Shape::print()` member function, as well as a pair of methods, `Shape_nshapes_get()` and `Shape_nshapes_set()`, to get and set the value of `Shape::nshapes`.

## 26 SWIG and Python

- [Overview](#)
- [Preliminaries](#)
  - ◆ [Running SWIG](#)
  - ◆ [Getting the right header files](#)
  - ◆ [Compiling a dynamic module](#)
  - ◆ [Using distutils](#)
  - ◆ [Static linking](#)
  - ◆ [Using your module](#)
  - ◆ [Compilation of C++ extensions](#)
  - ◆ [Compiling for 64-bit platforms](#)
  - ◆ [Building Python Extensions under Windows](#)
- [A tour of basic C/C++ wrapping](#)
  - ◆ [Modules](#)
  - ◆ [Functions](#)
  - ◆ [Global variables](#)
  - ◆ [Constants and enums](#)
  - ◆ [Pointers](#)
  - ◆ [Structures](#)
  - ◆ [C++ classes](#)
  - ◆ [C++ inheritance](#)
  - ◆ [Pointers, references, values, and arrays](#)
  - ◆ [C++ overloaded functions](#)
  - ◆ [C++ operators](#)
  - ◆ [C++ namespaces](#)
  - ◆ [C++ templates](#)
  - ◆ [C++ Smart Pointers](#)
- [Further details on the Python class interface](#)
  - ◆ [Proxy classes](#)
  - ◆ [Memory management](#)
  - ◆ [Python 2.2 and classic classes](#)
- [Cross language polymorphism \(experimental\)](#)
  - ◆ [Enabling directors](#)
  - ◆ [Director classes](#)
  - ◆ [Ownership and object destruction](#)
  - ◆ [Exception unrolling](#)
  - ◆ [Overhead and code bloat](#)
  - ◆ [Typemaps](#)
  - ◆ [Miscellaneous](#)
- [Common customization features](#)
  - ◆ [C/C++ helper functions](#)
  - ◆ [Adding additional Python code](#)
  - ◆ [Class extension with %extend](#)
  - ◆ [Exception handling with %exception](#)
- [Tips and techniques](#)
  - ◆ [Input and output parameters](#)
  - ◆ [Simple pointers](#)
  - ◆ [Unbounded C Arrays](#)
  - ◆ [String handling](#)
  - ◆ [Arrays](#)
  - ◆ [String arrays](#)
  - ◆ [STL wrappers](#)
- [Typemaps](#)
  - ◆ [What is a typemap?](#)

- ◆ [Python typemaps](#)
- ◆ [Typemap variables](#)
- ◆ [Useful Python Functions](#)
- [Typemap Examples](#)
  - ◆ [Converting Python list to a char \\*\\*](#)
  - ◆ [Expanding a Python object into multiple arguments](#)
  - ◆ [Using typemaps to return arguments](#)
  - ◆ [Mapping Python tuples into small arrays](#)
  - ◆ [Mapping sequences to C arrays](#)
  - ◆ [Pointer handling](#)
- [Docstring Features](#)
  - ◆ [Module docstring](#)
  - ◆ [%feature\("autodoc"\)](#)
    - ◇ [%feature\("autodoc", "0"\)](#)
    - ◇ [%feature\("autodoc", "1"\)](#)
    - ◇ [%feature\("autodoc", "docstring"\)](#)
  - ◆ [%feature\("docstring"\)](#)
- [Python Packages](#)

**Caution: This chapter is under repair!**

This chapter describes SWIG's support of Python. SWIG is compatible with most recent Python versions including Python 2.2 as well as older versions dating back to Python 1.5.2. For the best results, consider using Python 2.0 or newer.

This chapter covers most SWIG features, but certain low-level details are covered in less depth than in earlier chapters. At the very least, make sure you read the "[SWIG Basics](#)" chapter.

## 26.1 Overview

To build Python extension modules, SWIG uses a layered approach in which parts of the extension module are defined in C and other parts are defined in Python. The C layer contains low-level wrappers whereas Python code is used to define high-level features.

This layered approach recognizes the fact that certain aspects of extension building are better accomplished in each language (instead of trying to do everything in C or C++). Furthermore, by generating code in both languages, you get a lot more flexibility since you can enhance the extension module with support code in either language.

In describing the Python interface, this chapter starts by covering the basics of configuration, compiling, and installing Python modules. Next, the Python interface to common C and C++ programming features is described. Advanced customization features such as typemaps are then described followed by a discussion of low-level implementation details.

## 26.2 Preliminaries

### 26.2.1 Running SWIG

Suppose that you defined a SWIG module such as the following:

```
%module example
%{
#include "header.h"
%}
int fact(int n);
```

To build a Python module, run SWIG using the `-python` option :

```
$ swig -python example.i
```

If building a C++ extension, add the `-c++` option:

```
$ swig -c++ -python example.i
```

This creates two different files; a C/C++ source file `example_wrap.c` or `example_wrap.cxx` and a Python source file `example.py`. The generated C source file contains the low-level wrappers that need to be compiled and linked with the rest of your C/C++ application to create an extension module. The Python source file contains high-level support code. This is the file that you will import to use the module.

The name of the wrapper file is derived from the name of the input file. For example, if the input file is `example.i`, the name of the wrapper file is `example_wrap.c`. To change this, you can use the `-o` option. The name of the Python file is derived from the module name specified with `%module`. If the module name is `example`, then a file `example.py` is created.

## 26.2.2 Getting the right header files

In order to compile the C/C++ wrappers, the compiler needs the `Python.h` header file. This file is usually contained in a directory such as

```
/usr/local/include/python2.0
```

The exact location may vary on your machine, but the above location is typical. If you are not entirely sure where Python is installed, you can run Python to find out. For example:

```
$ python
Python 2.1.1 (#1, Jul 23 2001, 14:36:06)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> import sys
>>> print sys.prefix
/usr/local
>>>
```

## 26.2.3 Compiling a dynamic module

The preferred approach to building an extension module is to compile it into a shared object file or DLL. To do this, you need to compile your program using commands like this (shown for Linux):

```
$ swig -python example.i
$ gcc -c example.c
$ gcc -c example_wrap.c -I/usr/local/include/python2.0
$ gcc -shared example.o example_wrap.o -o _example.so
```

The exact commands for doing this vary from platform to platform. However, SWIG tries to guess the right options when it is installed. Therefore, you may want to start with one of the examples in the `SWIG/Examples/python` directory. If that doesn't work, you will need to read the man-pages for your compiler and linker to get the right set of options. You might also check the [SWIG Wiki](#) for additional information.

When linking the module, **the name of the output file has to match the name of the module prefixed by an underscore**. If the name of your module is `"example"`, then the name of the corresponding object file should be `"_example.so"` or `"_examplemodule.so"`. The name of the module is specified using the `%module` directive or the `-module` command line option.

**Compatibility Note:** In SWIG-1.3.13 and earlier releases, module names did not include the leading underscore. This is because modules were normally created as C-only extensions without the extra Python support file (instead, creating Python code was supported as an optional feature). This has been changed in SWIG-1.3.14 and is consistent with other Python extension modules. For example, the `socket` module actually consists of two files; `socket.py` and `_socket.so`. Many other built-in Python modules follow a similar convention.

## 26.2.4 Using distutils

## 26.2.5 Static linking

An alternative approach to dynamic linking is to rebuild the Python interpreter with your extension module added to it. In the past, this approach was sometimes necessary due to limitations in dynamic loading support on certain machines. However, the situation has improved greatly over the last few years and you should not consider this approach unless there is really no other option.

The usual procedure for adding a new module to Python involves finding the Python source, adding an entry to the `Modules/Setup` file, and rebuilding the interpreter using the Python Makefile. However, newer Python versions have changed the build process. You may need to edit the `'setup.py'` file in the Python distribution instead.

In earlier versions of SWIG, the `embed.i` library file could be used to rebuild the interpreter. For example:

```
%module example

extern int fact(int);
extern int mod(int, int);
extern double My_variable;

#include embed.i           // Include code for a static version of Python
```

The `embed.i` library file includes supporting code that contains everything needed to rebuild Python. To rebuild the interpreter, you simply do something like this:

```
$ swig -python example.i
$ gcc example.c example_wrap.c \
    -Xlinker -export-dynamic \
    -DHAVE_CONFIG_H -I/usr/local/include/python2.1 \
    -I/usr/local/lib/python2.1/config \
    -L/usr/local/lib/python2.1/config -lpython2.1 -lm -ldl \
    -o mypython
```

You will need to supply the same libraries that were used to build Python the first time. This may include system libraries such as `-lsocket`, `-lnsl`, and `-lpthread`. Assuming this actually works, the new version of Python should be identical to the default version except that your extension module will be a built-in part of the interpreter.

**Comment:** In practice, you should probably try to avoid static linking if possible. Some programmers may be inclined to use static linking in the interest of getting better performance. However, the performance gained by static linking tends to be rather minimal in most situations (and quite frankly not worth the extra hassle in the opinion of this author).

**Compatibility note:** The `embed.i` library file is deprecated and has not been maintained for several years. Even though it appears to "work" with Python 2.1, no future support is guaranteed. If using static linking, you might want to rely on a different approach (perhaps using distutils).

## 26.2.6 Using your module

To use your module, simply use the Python `import` statement. If all goes well, you will be able to this:

```
$ python
>>> import example
>>> example.fact(4)
24
>>>
```

A common error received by first-time users is the following:



```
>>> import example
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    File "example.py", line 2, in ?
      import _example
ImportError: No module named _example
```

If you get this message, it means that you either forgot to compile the wrapper code into an extension module or you didn't give the extension module the right name. Make sure that you compiled the wrappers into a module called `_example.so`. And don't forget the leading underscore (`_`).

Another possible error is the following:

```
>>> import example
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: dynamic module does not define init function (init_example)
>>>
```

This error is almost always caused when a bad name is given to the shared object file. For example, if you created a file `example.so` instead of `_example.so` you would get this error. Alternatively, this error could arise if the name of the module is inconsistent with the module name supplied with the `%module` directive. Double-check the interface to make sure the module name and the shared object filename match. Another possible cause of this error is forgetting to link the SWIG-generated wrapper code with the rest of your application when creating the extension module.

Another common error is something similar to the following:

```
Traceback (most recent call last):
  File "example.py", line 3, in ?
    import example
ImportError: ./_example.so: undefined symbol: fact
```

This error usually indicates that you forgot to include some object files or libraries in the linking of the shared library file. Make sure you compile both the SWIG wrapper file and your original program into a shared library file. Make sure you pass all of the required libraries to the linker.

Sometimes unresolved symbols occur because a wrapper has been created for a function that doesn't actually exist in a library. This usually occurs when a header file includes a declaration for a function that was never actually implemented or it was removed from a library without updating the header file. To fix this, you can either edit the SWIG input file to remove the offending declaration or you can use the `%ignore` directive to ignore the declaration.

Finally, suppose that your extension module is linked with another library like this:

```
$ gcc -shared example.o example_wrap.o -L/home/beazley/projects/lib -lfoo \
  -o _example.so
```

If the `foo` library is compiled as a shared library, you might encounter the following problem when you try to use your module:

```
>>> import example
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: libfoo.so: cannot open shared object file: No such file or directory
>>>
```

This error is generated because the dynamic linker can't locate the `libfoo.so` library. When shared libraries are loaded, the system normally only checks a few standard locations such as `/usr/lib` and `/usr/local/lib`. To fix this problem, there are several things you can do. First, you can recompile your extension module with extra path information. For example, on Linux you can do this:

```
$ gcc -shared example.o example_wrap.o -L/home/beazley/projects/lib -lfoo \
```

```
-Xlinker -rpath /home/beazley/projects/lib \
-o _example.so
```

Alternatively, you can set the `LD_LIBRARY_PATH` environment variable to include the directory with your shared libraries. If setting `LD_LIBRARY_PATH`, be aware that setting this variable can introduce a noticeable performance impact on all other applications that you run. To set it only for Python, you might want to do this instead:

```
$ env LD_LIBRARY_PATH=/home/beazley/projects/lib python
```

Finally, you can use a command such as `ldconfig` (Linux) or `crle` (Solaris) to add additional search paths to the default system configuration (this requires root access and you will need to read the man pages).

## 26.2.7 Compilation of C++ extensions

Compilation of C++ extensions has traditionally been a tricky problem. Since the Python interpreter is written in C, you need to take steps to make sure C++ is properly initialized and that modules are compiled correctly.

On most machines, C++ extension modules should be linked using the C++ compiler. For example:

```
% swig -c++ -python example.i
% g++ -c example.cxx
% g++ -c example_wrap.cxx -I/usr/local/include/python2.0
% g++ -shared example.o example_wrap.o -o _example.so
```

In addition to this, you may need to include additional library files to make it work. For example, if you are using the Sun C++ compiler on Solaris, you often need to add an extra library `-lCrun` like this:

```
% swig -c++ -python example.i
% CC -c example.cxx
% CC -c example_wrap.cxx -I/usr/local/include/python2.0
% CC -G example.o example_wrap.o -I/opt/SUNWsprow/lib -o _example.so -lCrun
```

Of course, the extra libraries to use are completely non-portable—you will probably need to do some experimentation.

Sometimes people have suggested that it is necessary to relink the Python interpreter using the C++ compiler to make C++ extension modules work. In the experience of this author, this has never actually appeared to be necessary. Relinking the interpreter with C++ really only includes the special run-time libraries described above—as long as you link your extension modules with these libraries, it should not be necessary to rebuild Python.

If you aren't entirely sure about the linking of a C++ extension, you might look at an existing C++ program. On many Unix machines, the `ldd` command will list library dependencies. This should give you some clues about what you might have to include when you link your extension module. For example:

```
$ ldd swig
    libstdc++-libc6.1-1.so.2 => /usr/lib/libstdc++-libc6.1-1.so.2 (0x40019000)
    libm.so.6 => /lib/libm.so.6 (0x4005b000)
    libc.so.6 => /lib/libc.so.6 (0x40077000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

As a final complication, a major weakness of C++ is that it does not define any sort of standard for binary linking of libraries. This means that C++ code compiled by different compilers will not link together properly as libraries nor is the memory layout of classes and data structures implemented in any kind of portable manner. In a monolithic C++ program, this problem may be unnoticed. However, in Python, it is possible for different extension modules to be compiled with different C++ compilers. As long as these modules are self-contained, this probably won't matter. However, if these modules start sharing data, you will need to take steps to avoid segmentation faults and other erratic program behavior. If working with lots of software components, you might want to investigate using a more formal standard such as COM.

## 26.2.8 Compiling for 64-bit platforms

On platforms that support 64-bit applications (Solaris, Irix, etc.), special care is required when building extension modules. On these machines, 64-bit applications are compiled and linked using a different set of compiler/linker options. In addition, it is not generally possible to mix 32-bit and 64-bit code together in the same application.

To utilize 64-bits, the Python executable will need to be recompiled as a 64-bit application. In addition, all libraries, wrapper code, and every other part of your application will need to be compiled for 64-bits. If you plan to use other third-party extension modules, they will also have to be recompiled as 64-bit extensions.

If you are wrapping commercial software for which you have no source code, you will be forced to use the same linking standard as used by that software. This may prevent the use of 64-bit extensions. It may also introduce problems on platforms that support more than one linking standard (e.g., `-o32` and `-n32` on Irix).

## 26.2.9 Building Python Extensions under Windows

Building a SWIG extension to Python under Windows is roughly similar to the process used with Unix. You will need to create a DLL that can be loaded into the interpreter. This section briefly describes the use of SWIG with Microsoft Visual C++. As a starting point, many of SWIG's examples include project files. You might want to take a quick look at these in addition to reading this section.

In Developer Studio, SWIG should be invoked as a custom build option. This is usually done as follows:

- Open up a new workspace and use the AppWizard to select a DLL project.
- Add both the SWIG interface file (the `.i` file), any supporting C files, and the name of the wrapper file that will be created by SWIG (ie. `example_wrap.c`). Note : If using C++, choose a different suffix for the wrapper file such as `example_wrap.cxx`. Don't worry if the wrapper file doesn't exist yet—Developer Studio keeps a reference to it.
- Select the SWIG interface file and go to the settings menu. Under settings, select the "Custom Build" option.
- Enter "SWIG" in the description field.
- Enter `swig -python -o $(ProjDir)\$(InputName)_wrap.c $(InputPath)` in the "Build command(s) field"
- Enter `$(ProjDir)\$(InputName)_wrap.c` in the "Output files(s) field".
- Next, select the settings for the entire project and go to "C++:Preprocessor". Add the include directories for your Python installation under "Additional include directories".
- Define the symbol `__WIN32__` under preprocessor options.
- Finally, select the settings for the entire project and go to "Link Options". Add the Python library file to your link libraries. For example `python21.lib`. Also, set the name of the output file to match the name of your Python module (ie. `_example.dll`).
- Build your project.

If all went well, SWIG will be automatically invoked whenever you build your project. Any changes made to the interface file will result in SWIG being automatically executed to produce a new version of the wrapper file.

To run your new Python extension, simply run Python and use the `import` command as normal. For example :

```
MSDOS > python
>>> import example
>>> print example.fact(4)
24
>>>
```

If you get an `ImportError` exception when importing the module, you may have forgotten to include additional library files when you built your module. If you get an access violation or some kind of general protection fault immediately upon import, you have a more serious problem. This is often caused by linking your extension module against the wrong set of Win32 debug or thread libraries. You will have to fiddle around with the build options of project to try and track this down.

Some users have reported success in building extension modules using Cygwin and other compilers. However, the problem of building usable DLLs with these compilers tends to be rather problematic. For the latest information, you may want to consult the [SWIG Wiki](#).

## 26.3 A tour of basic C/C++ wrapping

By default, SWIG tries to build a very natural Python interface to your C/C++ code. Functions are wrapped as functions, classes are wrapped as classes, and so forth. This section briefly covers the essential aspects of this wrapping.

### 26.3.1 Modules

The SWIG `%module` directive specifies the name of the Python module. If you specify `%module example`, then everything is wrapped into a Python `example` module. Underneath the covers, this module consists of a Python source file `example.py` and a low-level extension module `_example.so`. When choosing a module name, make sure you don't use the same name as a built-in Python command or standard module name.

### 26.3.2 Functions

Global functions are wrapped as new Python built-in functions. For example,

```
%module example
int fact(int n);
```

creates a built-in function `example.fact(n)` that works exactly like you think it does:

```
>>> import example
>>> print example.fact(4)
24
>>>
```

### 26.3.3 Global variables

C/C++ global variables are fully supported by SWIG. However, the underlying mechanism is somewhat different than you might expect due to the way that Python assignment works. When you type the following in Python

```
a = 3.4
```

"a" becomes a name for an object containing the value 3.4. If you later type

```
b = a
```

then "a" and "b" are both names for the object containing the value 3.4. Thus, there is only one object containing 3.4 and "a" and "b" are both names that refer to it. This is quite different than C where a variable name refers to a memory location in which a value is stored (and assignment copies data into that location). Because of this, there is no direct way to map variable assignment in C to variable assignment in Python.

To provide access to C global variables, SWIG creates a special object called `cvar` that is added to each SWIG generated module. Global variables are then accessed as attributes of this object. For example, consider this interface

```
// SWIG interface file with global variables
%module example
...
extern int My_variable;
extern double density;
...
```

Now look at the Python interface:

```
>>> import example
>>> # Print out value of a C global variable
>>> print example.cvar.My_variable
4
>>> # Set the value of a C global variable
>>> example.cvar.density = 0.8442
>>> # Use in a math operation
>>> example.cvar.density = example.cvar.density*1.10
```

If you make an error in variable assignment, you will receive an error message. For example:

```
>>> example.cvar.density = "Hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: C variable 'density (double )'
>>>
```

If a variable is declared as `const`, it is wrapped as a read-only variable. Attempts to modify its value will result in an error.

To make ordinary variables read-only, you can use the `%immutable` directive. For example:

```
%immutable;
extern char *path;
%mutable;
```

The `%immutable` directive stays in effect until it is explicitly disabled using `%mutable`.

If you just want to make a specific variable immutable, supply a declaration name. For example:

```
%immutable path;
...
extern char *path;      // Read-only (due to %immutable)
```

If you would like to access variables using a name other than `"cvar"`, it can be changed using the `-globals` option :

```
% swig -python -globals myvar example.i
```

Some care is in order when importing multiple SWIG modules. If you use the `"from <file> import *"` style of importing, you will get a name clash on the variable ``cvar'` and you will only be able to access global variables from the last module loaded. To prevent this, you might consider renaming `cvar` or making it private to the module by giving it a name that starts with a leading underscore. SWIG does not create `cvar` if there are no global variables in a module.

## 26.3.4 Constants and enums

C/C++ constants are installed as Python objects containing the appropriate value. To create a constant, use `#define`, `enum`, or the `%constant` directive. For example:

```
#define PI 3.14159
#define VERSION "1.0"

enum Beverage { ALE, LAGER, STOUT, PILSNER };

%constant int FOO = 42;
%constant const char *path = "/usr/local";
```

For enums, make sure that the definition of the enumeration actually appears in a header file or in the wrapper file somehow——if you just stick an enum in a SWIG interface without also telling the C compiler about it, the wrapper code won't compile.

Note: declarations declared as `const` are wrapped as read-only variables and will be accessed using the `cvar` object described in the previous section. They are not wrapped as constants. For further discussion about this, see the [SWIG Basics](#) chapter.

Constants are not guaranteed to remain constant in Python—the name of the constant could be accidentally reassigned to refer to some other object. Unfortunately, there is no easy way for SWIG to generate code that prevents this. You will just have to be careful.

### 26.3.5 Pointers

C/C++ pointers are fully supported by SWIG. Furthermore, SWIG has no problem working with incomplete type information. Here is a rather simple interface:

```
%module example

FILE *fopen(const char *filename, const char *mode);
int fputs(const char *, FILE *);
int fclose(FILE *);
```

When wrapped, you will be able to use the functions in a natural way from Python. For example:

```
>>> import example
>>> f = example.fopen("junk", "w")
>>> example.fputs("Hello World\n", f)
>>> example.fclose(f)
```

If this makes you uneasy, rest assured that there is no deep magic involved. Underneath the covers, pointers to C/C++ objects are simply represented as opaque values—normally an encoded character string like this:

```
>>> print f
_c0671108_p_FILE
>>>
```

This pointer value can be freely passed around to different C functions that expect to receive an object of type `FILE *`. The only thing you can't do is dereference the pointer from Python. Of course, that isn't much of a concern in this example.

As an alternative to strings, SWIG can encode pointers as a Python CObject type. CObjects are rarely discussed in most Python books or documentation. However, this is a special built-in type that can be used to hold raw C pointer values. Internally, a CObject is just a container that holds a raw `void *` along with some additional information such as a type-string.

If you want to use CObjects instead of strings, compile the SWIG wrapper code with the `-DSWIG_COBJECT_TYPES` option. For example:

```
% swig -python example.i
% gcc -c example.c
% gcc -c -DSWIG_COBJECT_TYPES example_wrap.c -I/usr/local/include/python2.0
% gcc -shared example.o example_wrap.o -o _example.so
```

The choice of whether or not to use strings or CObjects is mostly a matter of personal preference. There is no significant performance difference between using one type or the other (strings actually appear to be ever-so-slightly faster on the author's machine). Although CObjects feel more natural to some programmers, a disadvantage of this approach is that it makes debugging more difficult. For example, if you are using CObjects, you will get code that works like this:

```
>>> import example
>>> f = example.fopen("junk", "w")
>>> f
<PyCObject object at 0x80c5e60>
>>>
```

Notice how no clues regarding the actual type of `f` is shown. On the other hand, the string representation produces the following:

```
>>> f
'_c0671108_p_FILE'
>>>
```

For either pointer representation, the NULL pointer is represented by None.

As much as you might be inclined to modify a pointer value directly from Python, don't. The hexadecimal encoding is not necessarily the same as the logical memory address of the underlying object. Instead it is the raw byte encoding of the pointer value. The encoding will vary depending on the native byte-ordering of the platform (i.e., big-endian vs. little-endian). Similarly, don't try to manually cast a pointer to a new type by simply replacing the type-string. This may not work like you expect, it is particularly dangerous when casting C++ objects, and it won't work if you switch to a new pointer representation such as CObjects. If you need to cast a pointer or change its value, consider writing some helper functions instead. For example:

```
%inline %{
/* C-style cast */
Bar *FooToBar(Foo *f) {
    return (Bar *) f;
}

/* C++-style cast */
Foo *BarToFoo(Bar *b) {
    return dynamic_cast<Foo*>(b);
}

Foo *IncrFoo(Foo *f, int i) {
    return f+i;
}
%}
```

Also, if working with C++, you should always try to use the new C++ style casts. For example, in the above code, the C-style cast may return a bogus result whereas as the C++-style cast will return None if the conversion can't be performed.

### 26.3.6 Structures

If you wrap a C structure, it is wrapped by a Python class. This provides a very natural interface. For example,

```
struct Vector {
    double x,y,z;
};
```

is used as follows:

```
>>> v = example.Vector()
>>> v.x = 3.5
>>> v.y = 7.2
>>> print v.x, v.y, v.z
7.8 -4.5 0.0
>>>
```

Similar access is provided for unions and the data members of C++ classes.

If you print out the value of v in the above example, you will see something like this:

```
>>> print v
<C Vector instance at _18e31408_p_Vector>
```

This object is actually a Python instance that has been wrapped around a pointer to the low-level C structure. This instance doesn't actually do anything—it just serves as a proxy. The pointer to the C object can be found in the the `.this` attribute. For example:

```
>>> print v.this
_18e31408_p_Vector
>>>
```

Further details about the Python proxy class are covered a little later.

`const` members of a structure are read-only. Data members can also be forced to be read-only using the `%immutable` directive. For example:

```
struct Foo {
    ...
    %immutable;
    int x;          /* Read-only members */
    char *name;
    %mutable;
    ...
};
```

When `char *` members of a structure are wrapped, the contents are assumed to be dynamically allocated using `malloc` or `new` (depending on whether or not SWIG is run with the `-c++` option). When the structure member is set, the old contents will be released and a new value created. If this is not the behavior you want, you will have to use a `typemap` (described later).

If a structure contains arrays, access to those arrays is managed through pointers. For example, consider this:

```
struct Bar {
    int x[16];
};
```

If accessed in Python, you will see behavior like this:

```
>>> b = example.Bar()
>>> print b.x
_801861a4_p_int
>>>
```

This pointer can be passed around to functions that expect to receive an `int *` (just like C). You can also set the value of an array member using another pointer. For example:

```
>>> c = example.Bar()
>>> c.x = b.x          # Copy contents of b.x to c.x
```

For array assignment, SWIG copies the entire contents of the array starting with the data pointed to by `b.x`. In this example, 16 integers would be copied. Like C, SWIG makes no assumptions about bounds checking—if you pass a bad pointer, you may get a segmentation fault or access violation.

When a member of a structure is itself a structure, it is handled as a pointer. For example, suppose you have two structures like this:

```
struct Foo {
    int a;
};

struct Bar {
    Foo f;
};
```

Now, suppose that you access the `f` attribute of `Bar` like this:

```
>>> b = Bar()
>>> x = b.f
```

In this case, `x` is a pointer that points to the `Foo` that is inside `b`. This is the same value as generated by this C code:

```
Bar b;
Foo *x = &b->f;          /* Points inside b */
```



Because the pointer points inside the structure, you can modify the contents and everything works just like you would expect. For example:

```
>>> b = Bar()
>>> b.f.a = 3          # Modify attribute of structure member
>>> x = b.f
>>> x.a = 3           # Modifies the same structure
```

### 26.3.7 C++ classes

C++ classes are wrapped by Python classes as well. For example, if you have this class,

```
class List {
public:
    List();
    ~List();
    int  search(char *item);
    void insert(char *item);
    void remove(char *item);
    char *get(int n);
    int  length;
};
```

you can use it in Python like this:

```
>>> l = example.List()
>>> l.insert("Ale")
>>> l.insert("Stout")
>>> l.insert("Lager")
>>> l.get(1)
'Stout'
>>> print l.length
3
>>>
```

Class data members are accessed in the same manner as C structures.

Static class members present a special problem for Python. Prior to Python-2.2, Python classes had no support for static methods and no version of Python supports static member variables in a manner that SWIG can utilize. Therefore, SWIG generates wrappers that try to work around some of these issues. To illustrate, suppose you have a class like this:

```
class Spam {
public:
    static void foo();
    static int bar;
};
```

In Python, the static member can be access in three different ways:

```
>>> example.Spam.foo()    # Spam::foo()
>>> s = example.Spam()
>>> s.foo()               # Spam::foo() via an instance
>>> example.Spam.foo()    # Spam::foo(). Python-2.2 only
```

The first two methods of access are supported in all versions of Python. The last technique is only available in Python-2.2 and later versions.

Static member variables are currently accessed as global variables. This means, they are accessed through `cvar` like this:

```
>>> print example.cvar.Spam_bar
7
```

## 26.3.8 C++ inheritance

SWIG is fully aware of issues related to C++ inheritance. Therefore, if you have classes like this

```
class Foo {
...
};

class Bar : public Foo {
...
};
```

those classes are wrapped into a hierarchy of Python classes that reflect the same inheritance structure. All of the usual Python utility functions work normally:

```
>>> b = Bar()
>>> isinstance(b, Foo)
1
>>> issubclass(Bar, Foo)
1
>>> issubclass(Foo, Bar)
0
```

Furthermore, if you have functions like this

```
void spam(Foo *f);
```

then the function `spam( )` accepts `Foo *` or a pointer to any class derived from `Foo`.

It is safe to use multiple inheritance with SWIG.

## 26.3.9 Pointers, references, values, and arrays

In C++, there are many different ways a function might receive and manipulate objects. For example:

```
void spam1(Foo *x);      // Pass by pointer
void spam2(Foo &x);      // Pass by reference
void spam3(Foo x);       // Pass by value
void spam4(Foo x[]);     // Array of objects
```

In Python, there is no detailed distinction like this—specifically, there are only "objects". There are no pointers, references, arrays, and so forth. Because of this, SWIG unifies all of these types together in the wrapper code. For instance, if you actually had the above functions, it is perfectly legal to do this:

```
>>> f = Foo()           # Create a Foo
>>> spam1(f)            # Ok. Pointer
>>> spam2(f)            # Ok. Reference
>>> spam3(f)            # Ok. Value.
>>> spam4(f)            # Ok. Array (1 element)
```

Similar behavior occurs for return values. For example, if you had functions like this,

```
Foo *spam5();
Foo &spam6();
Foo spam7();
```

then all three functions will return a pointer to some `Foo` object. Since the third function (`spam7`) returns a value, newly allocated memory is used to hold the result and a pointer is returned (Python will release this memory when the return value is garbage collected).

## 26.3.10 C++ overloaded functions

C++ overloaded functions, methods, and constructors are mostly supported by SWIG. For example, if you have two functions like this:

```
void foo(int);
void foo(char *c);
```

You can use them in Python in a straightforward manner:

```
>>> foo(3)           # foo(int)
>>> foo("Hello")    # foo(char *c)
```

Similarly, if you have a class like this,

```
class Foo {
public:
    Foo();
    Foo(const Foo &);
    ...
};
```

you can write Python code like this:

```
>>> f = Foo()        # Create a Foo
>>> g = Foo(f)       # Copy f
```

Overloading support is not quite as flexible as in C++. Sometimes there are methods that SWIG can't disambiguate. For example:

```
void spam(int);
void spam(short);
```

or

```
void foo(Bar *b);
void foo(Bar &b);
```

If declarations such as these appear, you will get a warning message like this:

```
example.i:12: Warning(509): Overloaded spam(short) is shadowed by spam(int)
at example.i:11.
```

To fix this, you either need to ignore or rename one of the methods. For example:

```
%rename(spam_short) spam(short);
...
void spam(int);
void spam(short);    // Accessed as spam_short
```

or

```
%ignore spam(short);
...
void spam(int);
void spam(short);    // Ignored
```

SWIG resolves overloaded functions and methods using a disambiguation scheme that ranks and sorts declarations according to a set of type-precedence rules. The order in which declarations appear in the input does not matter except in situations where ambiguity arises—in this case, the first declaration takes precedence.

Please refer to the "SWIG and C++" chapter for more information about overloading.

### 26.3.11 C++ operators

Certain C++ overloaded operators can be handled automatically by SWIG. For example, consider a class like this:

```
class Complex {
private:
    double rpart, ipart;
public:
    Complex(double r = 0, double i = 0) : rpart(r), ipart(i) { }
    Complex(const Complex &c) : rpart(c.rpart), ipart(c.ipart) { }
    Complex &operator=(const Complex &c);
    Complex operator+(const Complex &c) const;
    Complex operator-(const Complex &c) const;
    Complex operator*(const Complex &c) const;
    Complex operator-() const;

    double re() const { return rpart; }
    double im() const { return ipart; }
};
```

When wrapped, it works like you expect:

```
>>> c = Complex(3,4)
>>> d = Complex(7,8)
>>> e = c + d
>>> e.re()
10.0
>>> e.im()
12.0
```

One restriction with operator overloading support is that SWIG is not able to fully handle operators that aren't defined as part of the class. For example, if you had code like this

```
class Complex {
...
friend Complex operator+(double, const Complex &c);
...
};
```

then SWIG doesn't know what to do with the friend function—in fact, it simply ignores it and issues a warning. You can still wrap the operator, but you may have to encapsulate it in a special function. For example:

```
%rename(Complex_add_dc) operator+(double, const Complex &);
...
Complex operator+(double, const Complex &c);
```

There are ways to make this operator appear as part of the class using the `%extend` directive. Keep reading.

Also, be aware that certain operators don't map cleanly to Python. For instance, overloaded assignment operators don't map to Python semantics and will be ignored.

### 26.3.12 C++ namespaces

SWIG is aware of C++ namespaces, but namespace names do not appear in the module nor do namespaces result in a module that is broken up into submodules or packages. For example, if you have a file like this,

```
%module example

namespace foo {
    int fact(int n);
```

```

    struct Vector {
        double x,y,z;
    };
};

```

it works in Python as follows:

```

>>> import example
>>> example.fact(3)
6
>>> v = example.Vector()
>>> v.x = 3.4
>>> print v.y
0.0
>>>

```

If your program has more than one namespace, name conflicts (if any) can be resolved using `%rename`. For example:

```

%rename(Bar_spam) Bar::spam;

namespace Foo {
    int spam();
}

namespace Bar {
    int spam();
}

```

If you have more than one namespace and you want to keep their symbols separate, consider wrapping them as separate SWIG modules. For example, make the module name the same as the namespace and create extension modules for each namespace separately. If your program utilizes thousands of small deeply nested namespaces each with identical symbol names, well, then you get what you deserve.

### 26.3.13 C++ templates

C++ templates don't present a huge problem for SWIG. However, in order to create wrappers, you have to tell SWIG to create wrappers for a particular template instantiation. To do this, you use the `%template` directive. For example:

```

%module example
%{
#include "pair.h"
%}

template<class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1&, const T2&);
    ~pair();
};

%template(pairii) pair<int,int>;

```

In Python:

```

>>> import example
>>> p = example.pairii(3,4)
>>> p.first
3
>>> p.second

```

Obviously, there is more to template wrapping than shown in this example. More details can be found in the [SWIG and C++](#) chapter. Some more complicated examples will appear later.

### 26.3.14 C++ Smart Pointers

In certain C++ programs, it is common to use classes that have been wrapped by so-called "smart pointers." Generally, this involves the use of a template class that implements `operator->()` like this:

```
template<class T> class SmartPtr {
    ...
    T *operator->();
    ...
}
```

Then, if you have a class like this,

```
class Foo {
public:
    int x;
    int bar();
};
```

A smart pointer would be used in C++ as follows:

```
SmartPtr<Foo> p = CreateFoo();    // Created somehow (not shown)
...
p->x = 3;                        // Foo::x
int y = p->bar();                // Foo::bar
```

To wrap this in Python, simply tell SWIG about the `SmartPtr` class and the low-level `Foo` object. Make sure you instantiate `SmartPtr` using `%template` if necessary. For example:

```
%module example
...
%template(SmartPtrFoo) SmartPtr<Foo>;
...
```

Now, in Python, everything should just "work":

```
>>> p = example.CreateFoo()      # Create a smart-pointer somehow
>>> p.x = 3                     # Foo::x
>>> p.bar()                     # Foo::bar
```

If you ever need to access the underlying pointer returned by `operator->()` itself, simply use the `__deref__()` method. For example:

```
>>> f = p.__deref__()           # Returns underlying Foo *
```

## 26.4 Further details on the Python class interface

In the previous section, a high-level view of Python wrapping was presented. A key component of this wrapping is that structures and classes are wrapped by Python proxy classes. This provides a very natural Python interface and allows SWIG to support a number of advanced features such as operator overloading. However, a number of low-level details were omitted. This section provides a brief overview of how the proxy classes work.

## 26.4.1 Proxy classes

In the ["SWIG basics"](#) and ["SWIG and C++"](#) chapters, details of low-level structure and class wrapping are described. To summarize those chapters, if you have a class like this

```
class Foo {
public:
    int x;
    int spam(int);
    ...
}
```

then SWIG transforms it into a set of low-level procedural wrappers. For example:

```
Foo *new_Foo() {
    return new Foo();
}
void delete_Foo(Foo *f) {
    delete f;
}
int Foo_x_get(Foo *f) {
    return f->x;
}
void Foo_x_set(Foo *f, int value) {
    f->x = value;
}
int Foo_spam(Foo *f, int arg1) {
    return f->spam(arg1);
}
```

These wrappers can be found in the low-level extension module (e.g., `_example`).

Using these wrappers, SWIG generates a high-level Python proxy class (also known as a shadow class) like this (shown for Python 2.2):

```
import _example

class Foo(object):
    def __init__(self):
        self.this = _example.new_Foo()
        self.thisown = 1
    def __del__(self):
        if self.thisown:
            _example.delete_Foo(self.this)
    def spam(self, arg1):
        return _example.Foo_spam(self.this, arg1)
    x = property(_example.Foo_x_get, _example.Foo_x_set)
```

This class merely holds a pointer to the underlying C++ object (`.this`) and dispatches methods and member variable access to that object using the low-level accessor functions. From a user's point of view, it makes the class work normally:

```
>>> f = example.Foo()
>>> f.x = 3
>>> y = f.spam(5)
```

The fact that the class has been wrapped by a real Python class offers certain advantages. For instance, you can attach new Python methods to the class and you can even inherit from it (something not supported by Python built-in types until Python 2.2).

## 26.4.2 Memory management

Associated with proxy object, is an ownership flag `.thisown`. The value of this flag determines who is responsible for deleting the underlying C++ object. If set to 1, the Python interpreter will destroy the C++ object when the proxy class is garbage collected. If set to 0 (or if the attribute is missing), then the destruction of the proxy class has no effect on the C++ object.

When an object is created by a constructor or returned by value, Python automatically takes ownership of the result. For example:

```
class Foo {
public:
    Foo();
    Foo bar();
};
```

In Python:

```
>>> f = Foo()
>>> f.thisown
1
>>> g = f.bar()
>>> g.thisown
1
```

On the other hand, when pointers are returned to Python, there is often no way to know where they came from. Therefore, the ownership is set to zero. For example:

```
class Foo {
public:
    ...
    Foo *spam();
    ...
};

>>> f = Foo()
>>> s = f.spam()
>>> print s.thisown
0
>>>
```

This behavior is especially important for classes that act as containers. For example, if a method returns a pointer to an object that is contained inside another object, you definitely don't want Python to assume ownership and destroy it!

Related to containers, ownership issues can arise whenever an object is assigned to a member or global variable. For example, consider this interface:

```
%module example

struct Foo {
    int value;
    Foo *next;
};

Foo *head = 0;
```

When wrapped in Python, careful observation will reveal that ownership changes whenever an object is assigned to a global variable. For example:

```
>>> f = example.Foo()
>>> f.thisown
1
>>> example.cvar.head = f
>>> f.thisown
0
>>>
```

In this case, C is now holding a reference to the object—you probably don't want Python to destroy it. Similarly, this occurs for members. For example:

```
>>> f = example.Foo()
```



```
>>> g = example.Foo()
>>> f.thisown
1
>>> g.thisown
1
>>> f.next = g
>>> g.thisown
0
>>>
```

For the most part, memory management issues remain hidden. However, there are occasionally situations where you might have to manually change the ownership of an object. For instance, consider code like this:

```
class Node {
    Object *value;
public:
    void set_value(Object *v) { value = v; }
    ...
};
```

Now, consider the following Python code:

```
>>> v = Object()           # Create an object
>>> n = Node()             # Create a node
>>> n.set_value(v)         # Set value
>>> v.thisown
1
>>> del v
```

In this case, the object `n` is holding a reference to `v` internally. However, SWIG has no way to know that this has occurred. Therefore, Python still thinks that it has ownership of the object. Should the proxy object be destroyed, then the C++ destructor will be invoked and `n` will be holding a stale-pointer. If you're lucky, you will only get a segmentation fault.

To work around this, it is always possible to flip the ownership flag. For example,

```
>>> v.thisown = 0
```

It is also possible to deal with situations like this using `typemaps`—an advanced topic discussed later.

### 26.4.3 Python 2.2 and classic classes

SWIG makes every attempt to preserve backwards compatibility with older versions of Python to the extent that it is possible. However, in Python-2.2, an entirely new type of class system was introduced. This new-style class system offers many enhancements including static member functions, properties (managed attributes), and class methods. Details about all of these changes can be found on [www.python.org](http://www.python.org) and is not repeated here.

To address differences between Python versions, SWIG currently emits dual-mode proxy class wrappers. In Python-2.2 and newer releases, these wrappers encapsulate C++ objects in new-style classes that take advantage of new features (static methods and properties). However, if these very same wrappers are imported into an older version of Python, old-style classes are used instead.

This dual-nature of the wrapper code means that you can create extension modules with SWIG and those modules will work with all versions of Python ranging from Python-1.4 to the very latest release. Moreover, the wrappers take advantage of Python-2.2 features when available.

For the most part, the interface presented to users is the same regardless of what version of Python is used. The only incompatibility lies in the handling of static member functions. In Python-2.2, they can be accessed via the class itself. In Python-2.1 and earlier, they have to be accessed as a global function or through an instance (see the earlier section).

## 26.5 Cross language polymorphism (experimental)

Proxy classes provide a more natural, object-oriented way to access extension classes. As described above, each proxy instance has an associated C++ instance, and method calls to the proxy are passed to the C++ instance transparently via C wrapper functions.

This arrangement is asymmetric in the sense that no corresponding mechanism exists to pass method calls down the inheritance chain from C++ to Python. In particular, if a C++ class has been extended in Python (by extending the proxy class), these extensions will not be visible from C++ code. Virtual method calls from C++ are thus not able access the lowest implementation in the inheritance chain.

Changes have been made to SWIG 1.3.18 to address this problem and make the relationship between C++ classes and proxy classes more symmetric. To achieve this goal, new classes called directors are introduced at the bottom of the C++ inheritance chain. The job of the directors is to route method calls correctly, either to C++ implementations higher in the inheritance chain or to Python implementations lower in the inheritance chain. The upshot is that C++ classes can be extended in Python and from C++ these extensions look exactly like native C++ classes. Neither C++ code nor Python code needs to know where a particular method is implemented: the combination of proxy classes, director classes, and C wrapper functions takes care of all the cross-language method routing transparently.

### 26.5.1 Enabling directors

The director feature is disabled by default. To use directors you must make two changes to the interface file. First, add the "directors" option to the %module directive, like this:

```
%module(directors="1") modulename
```

Without this option no director code will be generated. Second, you must use the %feature("director") directive to tell SWIG which classes and methods should get directors. The %feature directive can be applied globally, to specific classes, and to specific methods, like this:

```
// generate directors for all classes that have virtual methods
%feature("director");

// generate directors for all virtual methods in class Foo
%feature("director") Foo;

// generate a director for just Foo::bar()
%feature("director") Foo::bar;
```

You can use the %feature("nodirector") directive to turn off directors for specific classes or methods. So for example,

```
%feature("director") Foo;
%feature("nodirector") Foo::bar;
```

will generate directors for all virtual methods of class Foo except bar().

Directors can also be generated implicitly through inheritance. In the following, class Bar will get a director class that handles the methods one() and two() (but not three()):

```
%feature("director") Foo;
class Foo {
public:
    virtual void one();
    virtual void two();
};

class Bar: public Foo {
public:
    virtual void three();
```

```
};
```

## 26.5.2 Director classes

For each class that has directors enabled, SWIG generates a new class that derives from both the class in question and a special `Swig::Director` class. These new classes, referred to as director classes, can be loosely thought of as the C++ equivalent of the Python proxy classes. The director classes store a pointer to their underlying Python object and handle various issues related to object ownership. Indeed, this is quite similar to the "this" and "thisown" members of the Python proxy classes.

For simplicity let's ignore the `Swig::Director` class and refer to the original C++ class as the director's base class. By default, a director class extends all virtual methods in the inheritance chain of its base class (see the preceding section for how to modify this behavior). Thus all virtual method calls, whether they originate in C++ or in Python via proxy classes, eventually end up in the implementation in the director class. The job of the director methods is to route these method calls to the appropriate place in the inheritance chain. By "appropriate place" we mean the method that would have been called if the C++ base class and its extensions in Python were seamlessly integrated. That seamless integration is exactly what the director classes provide, transparently skipping over all the messy extension API glue that binds the two languages together.

In reality, the "appropriate place" is one of only two possibilities: C++ or Python. Once this decision is made, the rest is fairly easy. If the correct implementation is in C++, then the lowest implementation of the method in the C++ inheritance chain is called explicitly. If the correct implementation is in Python, the Python API is used to call the method of the underlying Python object (after which the usual virtual method resolution in Python automatically finds the right implementation).

Now how does the director decide which language should handle the method call? The basic rule is to handle the method in Python, unless there's a good reason not to. The reason for this is simple: Python has the most "extended" implementation of the method. This assertion is guaranteed, since at a minimum the Python proxy class implements the method. If the method in question has been extended by a class derived from the proxy class, that extended implementation will execute exactly as it should. If not, the proxy class will route the method call into a C wrapper function, expecting that the method will be resolved in C++. The wrapper will call the virtual method of the C++ instance, and since the director extends this the call will end up right back in the director method. Now comes the "good reason not to" part. If the director method were to blindly call the Python method again, it would get stuck in an infinite loop. We avoid this situation by adding special code to the C wrapper function that tells the director method to not do this. The C wrapper function compares the pointer to the Python object that called the wrapper function to the pointer stored by the director. If these are the same, then the C wrapper function tells the director to resolve the method by calling up the C++ inheritance chain, preventing an infinite loop.

One more point needs to be made about the relationship between director classes and proxy classes. When a proxy class instance is created in Python, SWIG creates an instance of the original C++ class and assigns it to `.this`. This is exactly what happens without directors and is true even if directors are enabled for the particular class in question. When a class *derived* from a proxy class is created, however, SWIG then creates an instance of the corresponding C++ director class. The reason for this difference is that user-defined subclasses may override or extend methods of the original class, so the director class is needed to route calls to these methods correctly. For unmodified proxy classes, all methods are ultimately implemented in C++ so there is no need for the extra overhead involved with routing the calls through Python.

## 26.5.3 Ownership and object destruction

Memory management issues are slightly more complicated with directors than for proxy classes alone. Python instances hold a pointer to the associated C++ director object, and the director in turn holds a pointer back to the Python object. By default, proxy classes own their C++ director object and take care of deleting it when they are garbage collected.

This relationship can be reversed by calling the special `__disown__()` method of the proxy class. After calling this method, the `.thisown` flag is set to zero, and the director class increments the reference count of the Python object. When the director class is deleted it decrements the reference count. Assuming no outstanding references to the Python object remain, the Python object will be destroyed at the same time. This is a good thing, since directors and proxies refer to each other and so must be created and destroyed together. Destroying one without destroying the other will likely cause your program to segfault.

To help ensure that no references to the Python object remain after calling `__disown__()`, this method returns a weak reference to the Python object. Weak references are only available in Python versions 2.1 and higher, so for older versions you

must explicitly delete all references. Here is an example:

```
class Foo {
public:
    ...
};
class FooContainer {
public:
    void addFoo(Foo *);
    ...
};

>>> c = FooContainer()
>>> a = Foo().__disown().__
>>> c.addFoo(a)
>>> b = Foo()
>>> b = b.__disown().__
>>> c.addFoo(b)
>>> c.addFoo(Foo().__disown().__)
```

In this example, we are assuming that FooContainer will take care of deleting all the Foo pointers it contains at some point. Note that no hard references to the Foo objects remain in Python.

### 26.5.4 Exception unrolling

With directors routing method calls to Python, and proxies routing them to C++, the handling of exceptions is an important concern. By default, the directors ignore exceptions that occur during method calls that are resolved in Python. To handle such exceptions correctly, it is necessary to temporarily translate them into C++ exceptions. This can be done with the `%feature("director:except")` directive. The following code should suffice in most cases:

```
%feature("director:except") {
    if ($error != NULL) {
        throw Swig::DirectorMethodException();
    }
}
```

This code will check the Python error state after each method call from a director into Python, and throw a C++ exception if an error occurred. This exception can be caught in C++ to implement an error handler. Currently no information about the Python error is stored in the `Swig::DirectorMethodException` object, but this will likely change in the future.

It may be the case that a method call originates in Python, travels up to C++ through a proxy class, and then back into Python via a director method. If an exception occurs in Python at this point, it would be nice for that exception to find its way back to the original caller. This can be done by combining a normal `%exception` directive with the `director:except` handler shown above. Here is an example of a suitable exception handler:

```
%exception {
    try { $action }
    catch (Swig::DirectorException &e) { SWIG_fail; }
}
```

The class `Swig::DirectorException` used in this example is actually a base class of `Swig::DirectorMethodException`, so it will trap this exception. Because the Python error state is still set when `Swig::DirectorMethodException` is thrown, Python will register the exception as soon as the C wrapper function returns.

### 26.5.5 Overhead and code bloat

Enabling directors for a class will generate a new director method for every virtual method in the class' inheritance chain. This alone can generate a lot of code bloat for large hierarchies. Method arguments that require complex conversions to and from target language types can result in large director methods. For this reason it is recommended that you selectively enable directors only for specific classes that are likely to be extended in Python and used in C++.

Compared to classes that do not use directors, the call routing in the director methods does add some overhead. In particular, at least one dynamic cast and one extra function call occur per method call from Python. Relative to the speed of Python execution this is probably completely negligible. For worst case routing, a method call that ultimately resolves in C++ may take one extra detour through Python in order to ensure that the method does not have an extended Python implementation. This could result in a noticeable overhead in some cases.

Although directors make it natural to mix native C++ objects with Python objects (as director objects) via a common base class pointer, one should be aware of the obvious fact that method calls to Python objects will be much slower than calls to C++ objects. This situation can be optimized by selectively enabling director methods (using the `%feature` directive) for only those methods that are likely to be extended in Python.

## 26.5.6 Typemaps

Typemaps for input and output of most of the basic types from director classes have been written. These are roughly the reverse of the usual input and output typemaps used by the wrapper code. The typemap operation names are 'directorin', 'directorout', and 'directorargout'. The director code does not use any of the other kinds of typemaps yet. It is not clear at this point which kinds are appropriate and need to be supported.

Typemaps for STL classes are under construction. So far there is support for `std::string`, `std::vector`, and `std::complex`, although there's no guarantee these are fully functional yet.

## 26.5.7 Miscellaneous

## 26.6 Common customization features

The last section presented the absolute basics of C/C++ wrapping. If you do nothing but feed SWIG a header file, you will get an interface that mimics the behavior described. However, sometimes this isn't enough to produce a nice module. Certain types of functionality might be missing or the interface to certain functions might be awkward. This section describes some common SWIG features that are used to improve your the interface to an extension module.

### 26.6.1 C/C++ helper functions

Sometimes when you create a module, it is missing certain bits of functionality. For example, if you had a function like this

```
void set_transform(Image *im, double m[4][4]);
```

it would be accessible from Python, but there may be no easy way to call it. For example, you might get errors like this:

```
>>> a = [
...     [1,0,0,0],
...     [0,1,0,0],
...     [0,0,1,0],
...     [0,0,0,1]]
>>> set_transform(im,a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Type error. Expected _p_a_4_double
```

The problem here is that there is no easy way to construct and manipulate a suitable `double [4][4]` value to use. To fix this, you can write some extra C helper functions. Just use the `%inline` directive. For example:

```
%inline %{
/* Note: double[4][4] is equivalent to a pointer to an array double (*)[4] */
double (*new_mat44())[4] {
    return (double (*)[4]) malloc(16*sizeof(double));
}
void free_mat44(double (*x)[4]) {
    free(x);
}
```

```

}
void mat44_set(double x[4][4], int i, int j, double v) {
    x[i][j] = v;
}
double mat44_get(double x[4][4], int i, int j) {
    return x[i][j];
}
%}

```

From Python, you could then write code like this:

```

>>> a = new_mat44()
>>> mat44_set(a,0,0,1.0)
>>> mat44_set(a,1,1,1.0)
>>> mat44_set(a,2,2,1.0)
...
>>> set_transform(im,a)
>>>

```

Admittedly, this is not the most elegant looking approach. However, it works and it wasn't too hard to implement. It is possible to clean this up using Python code, typemaps, and other customization features as covered in later sections.

## 26.6.2 Adding additional Python code

If writing support code in C isn't enough, it is also possible to write code in Python. This code gets inserted in to the .py file created by SWIG. One use of Python code might be to supply a high-level interface to certain functions. For example:

```

void set_transform(Image *im, double x[4][4]);

...
/* Rewrite the high level interface to set_transform */
%pythoncode %{
def set_transform(im,x):
    a = new_mat44()
    for i in range(4):
        for j in range(4):
            mat44_set(a,i,j,x[i][j])
    _example.set_transform(im,a)
    free_mat44(a)
%}

```

In this example, `set_transform()` provides a high-level Python interface built on top of low-level helper functions. For example, this code now seems to work:

```

>>> a = [
...     [1,0,0,0],
...     [0,1,0,0],
...     [0,0,1,0],
...     [0,0,0,1]]
>>> set_transform(im,a)
>>>

```

Admittedly, this whole scheme for wrapping the two-dimension array argument is rather ad-hoc. Besides, shouldn't a Python list or a Numeric Python array just work normally? We'll get to those examples soon enough. For now, think of this example as an illustration of what can be done without having to rely on any of the more advanced customization features.

## 26.6.3 Class extension with %extend

One of the more interesting features of SWIG is that it can extend structures and classes with new methods—at least in the Python interface. Here is a simple example:

```
%module example
```

```
%{
#include "someheader.h"
%}

struct Vector {
    double x,y,z;
};

%extend Vector {
    char *__str__() {
        static char tmp[1024];
        sprintf(tmp, "Vector(%g,%g,%g)", self->x, self->y, self->z);
        return tmp;
    }
    Vector(double x, double y, double z) {
        Vector *v = (Vector *) malloc(sizeof(Vector));
        v->x = x;
        v->y = y;
        v->z = z;
        return v;
    }
};
```

Now, in Python

```
>>> v = example.Vector(2,3,4)
>>> print v
Vector(2,3,4)
>>>
```

`%extend` can be used for many more tasks than this. For example, if you wanted to overload a Python operator, you might do this:

```
%extend Vector {
    Vector __add__(Vector *other) {
        Vector v;
        v.x = self->x + other->x;
        v.y = self->y + other->y;
        v.z = self->z + other->z;
        return v;
    }
};
```

Use it like this:

```
>>> import example
>>> v = example.Vector(2,3,4)
>>> w = example.Vector(10,11,12)
>>> print v+w
Vector(12,14,16)
>>>
```

`%extend` works with both C and C++ code. It does not modify the underlying object in any way—the extensions only show up in the Python interface.

## 26.6.4 Exception handling with `%exception`

If a C or C++ function throws an error, you may want to convert that error into a Python exception. To do this, you can use the `%exception` directive. `%exception` simply lets you rewrite part of the generated wrapper code to include an error check.

In C, a function often indicates an error by returning a status code (a negative number or a NULL pointer perhaps). Here is a simple example of how you might handle that:

```
%exception malloc {
    $action
    if (!result) {
        PyErr_SetString(PyExc_MemoryError, "Not enough memory");
        return NULL;
    }
}
void *malloc(size_t nbytes);
```

In Python,

```
>>> a = example.malloc(2000000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
MemoryError: Not enough memory
>>>
```

If a library provides some kind of general error handling framework, you can also use that. For example:

```
%exception {
    $action
    if (err_occurred()) {
        PyErr_SetString(PyExc_RuntimeError, err_message());
        return NULL;
    }
}
```

No declaration name is given to `%exception`, it is applied to all wrapper functions.

C++ exceptions are also easy to handle. For example, you can write code like this:

```
%exception getitem {
    try {
        $action
    } catch (std::out_of_range &e) {
        PyErr_SetString(PyExc_IndexError, const_cast<char*>(e.what()));
        return NULL;
    }
}

class Base {
public:
    Foo *getitem(int index);      // Exception handled added
    ...
};
```

When raising a Python exception from C, use the `PyErr_SetString()` function as shown above. The following exception types can be used as the first argument.

```
PyExc_ArithmeticError
PyExc_AssertionError
PyExc_AttributeError
PyExc_EnvironmentError
PyExc_EOFError
PyExc_Exception
PyExc_FloatingPointError
PyExc_ImportError
PyExc_IndexError
PyExc_IOError
PyExc_KeyError
PyExc_KeyboardInterrupt
PyExc_LookupError
PyExc_MemoryError
PyExc_NameError
PyExc_NotImplementedError
```



```

PyExc_OSError
PyExc_OverflowError
PyExc_RuntimeError
PyExc_StandardError
PyExc_SyntaxError
PyExc_SystemError
PyExc_TypeError
PyExc_UnicodeError
PyExc_ValueError
PyExc_ZeroDivisionError

```

The language-independent `exception.i` library file can also be used to raise exceptions. See the [SWIG Library](#) chapter.

## 26.7 Tips and techniques

Although SWIG is largely automatic, there are certain types of wrapping problems that require additional user input. Examples include dealing with output parameters, strings, binary data, and arrays. This chapter discusses the common techniques for solving these problems.

### 26.7.1 Input and output parameters

A common problem in some C programs is handling parameters passed as simple pointers. For example:

```

void add(int x, int y, int *result) {
    *result = x + y;
}

```

or perhaps

```

int sub(int *x, int *y) {
    return *x-*y;
}

```

The easiest way to handle these situations is to use the `typemaps.i` file. For example:

```

%module example
#include "typemaps.i"

void add(int, int, int *OUTPUT);
int sub(int *INPUT, int *INPUT);

```

In Python, this allows you to pass simple values. For example:

```

>>> a = add(3,4)
>>> print a
7
>>> b = sub(7,4)
>>> print b
3
>>>

```

Notice how the `INPUT` parameters allow integer values to be passed instead of pointers and how the `OUTPUT` parameter creates a return result.

If you don't want to use the names `INPUT` or `OUTPUT`, use the `%apply` directive. For example:

```

%module example
#include "typemaps.i"

%apply int *OUTPUT { int *result };
%apply int *INPUT { int *x, int *y};

```

```
void add(int x, int y, int *result);
int sub(int *x, int *y);
```

If a function mutates one of its parameters like this,

```
void negate(int *x) {
    *x = -(*x);
}
```

you can use INOUT like this:

```
%include "typemaps.i"
...
void negate(int *INOUT);
```

In Python, a mutated parameter shows up as a return value. For example:

```
>>> a = negate(3)
>>> print a
-3
>>>
```

Note: Since most primitive Python objects are immutable, it is not possible to perform in-place modification of a Python object passed as a parameter.

The most common use of these special typemap rules is to handle functions that return more than one value. For example, sometimes a function returns a result as well as a special error code:

```
/* send message, return number of bytes sent, along with success code */
int send_message(char *text, int len, int *success);
```

To wrap such a function, simply use the OUTPUT rule above. For example:

```
%module example
#include "typemaps.i"
%apply int *OUTPUT { int *success };
...
int send_message(char *text, int *success);
```

When used in Python, the function will return multiple values.

```
bytes, success = send_message("Hello World")
if not success:
    print "Whoa!"
else:
    print "Sent", bytes
```

Another common use of multiple return values are in query functions. For example:

```
void get_dimensions(Matrix *m, int *rows, int *columns);
```

To wrap this, you might use the following:

```
%module example
#include "typemaps.i"
%apply int *OUTPUT { int *rows, int *columns };
...
void get_dimensions(Matrix *m, int *rows, *columns);
```

Now, in Python:

```
>>> r,c = get_dimensions(m)
```

Be aware that the primary purpose of the `typemaps.i` file is to support primitive datatypes. Writing a function like this

```
void foo(Bar *OUTPUT);
```

may not have the intended effect since `typemaps.i` does not define an `OUTPUT` rule for `Bar`.

## 26.7.2 Simple pointers

If you must work with simple pointers such as `int *` or `double *` and you don't want to use `typemaps.i`, consider using the `cpointer.i` library file. For example:

```
%module example
#include "cpointer.i"

extern void add(int x, int y, int *result);
%pointer_functions(int, intp);
```

The `%pointer_functions(type, name)` macro generates five helper functions that can be used to create, destroy, copy, assign, and dereference a pointer. In this case, the functions are as follows:

```
int *new_intp();
int *copy_intp(int *x);
void delete_intp(int *x);
void intp_assign(int *x, int value);
int intp_value(int *x);
```

In Python, you would use the functions like this:

```
>>> result = new_intp()
>>> print result
_108fea8_p_int
>>> add(3,4,result)
>>> print intp_value(result)
7
>>>
```

If you replace `%pointer_functions()` by `%pointer_class(type, name)`, the interface is more class-like.

```
>>> result = intp()
>>> add(3,4,result)
>>> print result.value()
7
```

See the [SWIG Library](#) chapter for further details.

## 26.7.3 Unbounded C Arrays

Sometimes a C function expects an array to be passed as a pointer. For example,

```
int sumitems(int *first, int nitems) {
    int i, sum = 0;
    for (i = 0; i < nitems; i++) {
        sum += first[i];
    }
    return sum;
}
```

To wrap this into Python, you need to pass an array pointer as the first argument. A simple way to do this is to use the `carrays.i` library file. For example:

```
%include "carrays.i"
%array_class(int, intArray);
```

The `%array_class(type, name)` macro creates wrappers for an unbounded array object that can be passed around as a simple pointer like `int *` or `double *`. For instance, you will be able to do this in Python:

```
>>> a = intArray(10000000)           # Array of 10-million integers
>>> for i in xrange(10000):          # Set some values
...     a[i] = i
>>> sumitems(a,10000)
49995000
>>>
```

The array "object" created by `%array_class()` does not encapsulate pointers inside a special array object. In fact, there is no bounds checking or safety of any kind (just like in C). Because of this, the arrays created by this library are extremely low-level indeed. You can't iterate over them nor can you even query their length. In fact, any valid memory address can be accessed if you want (negative indices, indices beyond the end of the array, etc.). Needless to say, this approach is not going to suit all applications. On the other hand, this low-level approach is extremely efficient and well suited for applications in which you need to create buffers, package binary data, etc.

## 26.7.4 String handling

If a C function has an argument of `char *`, then a Python string can be passed as input. For example:

```
// C
void foo(char *s);

# Python
>>> foo("Hello")
```

When a Python string is passed as a parameter, the C function receives a pointer to the raw data contained in the string. Since Python strings are immutable, it is illegal for your program to change the value. In fact, doing so will probably crash the Python interpreter.

If your program modifies the input parameter or uses it to return data, consider using the `cstring.i` library file described in the [SWIG Library](#) chapter.

When functions return a `char *`, it is assumed to be a NULL-terminated string. Data is copied into a new Python string and returned.

If your program needs to work with binary data, you can use a typemap to expand a Python string into a pointer/length argument pair. As luck would have it, just such a typemap is already defined. Just do this:

```
%apply (char *STRING, int LENGTH) { (char *data, int size) };
...
int parity(char *data, int size, int initial);
```

Now in Python:

```
>>> parity("e\x09ffss\x00\x00\x01\nx", 0)
```

If you need to return binary data, you might use the `cstring.i` library file. The `cdata.i` library can also be used to extra binary data from arbitrary pointers.

## 26.7.5 Arrays

## 26.7.6 String arrays

## 26.7.7 STL wrappers

# 26.8 Typemaps

This section describes how you can modify SWIG's default wrapping behavior for various C/C++ datatypes using the `%typemap` directive. This is an advanced topic that assumes familiarity with the Python C API as well as the material in the ["Typemaps"](#) chapter.

Before proceeding, it should be stressed that typemaps are not a required part of using SWIG—the default wrapping behavior is enough in most cases. Typemaps are only used if you want to change some aspect of the primitive C–Python interface or if you want to elevate your guru status.

## 26.8.1 What is a typemap?

A typemap is nothing more than a code generation rule that is attached to a specific C datatype. For example, to convert integers from Python to C, you might define a typemap like this:

```
%module example

%typemap(in) int {
    $1 = (int) PyLong_AsLong($input);
    printf("Received an integer : %d\n", $1);
}
extern int fact(int n);
```

Typemaps are always associated with some specific aspect of code generation. In this case, the "in" method refers to the conversion of input arguments to C/C++. The datatype `int` is the datatype to which the typemap will be applied. The supplied C code is used to convert values. In this code a number of special variable prefaced by a `$` are used. The `$1` variable is placeholder for a local variable of type `int`. The `$input` variable is the input object of type `PyObject *`.

When this example is compiled into a Python module, it operates as follows:

```
>>> from example import *
>>> fact(6)
Received an integer : 6
720
```

In this example, the typemap is applied to all occurrences of the `int` datatype. You can refine this by supplying an optional parameter name. For example:

```
%module example

%typemap(in) int nonnegative {
    $1 = (int) PyLong_AsLong($input);
    if ($1 < 0) {
        PyErr_SetString(PyExc_ValueError, "Expected a nonnegative value.");
        return NULL;
    }
}
extern int fact(int nonnegative);
```

In this case, the typemap code is only attached to arguments that exactly match `int nonnegative`.

The application of a typemap to specific datatypes and argument names involves more than simple text-matching—typemaps are fully integrated into the SWIG C++ type-system. When you define a typemap for `int`, that typemap applies to `int` and qualified variations such as `const int`. In addition, the typemap system follows `typedef` declarations. For example:

```
%typemap(in) int n {
    $1 = (int) PyLong_AsLong($input);
    printf("n = %d\n", $1);
}
typedef int Integer;
extern int fact(Integer n);    // Above typemap is applied
```

Typemaps can also be defined for groups of consecutive arguments. For example:

```
%typemap(in) (char *str, int len) {
    $1 = PyString_AsString($input);
    $2 = PyString_Size($input);
};

int count(char c, char *str, int len);
```

When a multi-argument typemap is defined, the arguments are always handled as a single Python object. This allows the function to be used like this (notice how the length parameter is omitted):

```
>>> example.count('e', 'Hello World')
1
>>>
```

## 26.8.2 Python typemaps

The previous section illustrated an "in" typemap for converting Python objects to C. A variety of different typemap methods are defined by the Python module. For example, to convert a C integer back into a Python object, you might define an "out" typemap like this:

```
%typemap(out) int {
    $result = PyInt_FromLong((long) $1);
}
```

A detailed list of available methods can be found in the "[Typemaps](#)" chapter. However, the best source of typemap information (and examples) is probably the Python module itself. In fact, all of SWIG's default type handling is defined by typemaps. You can view these typemaps by looking at the `python.swg` file in the SWIG library. Just issue these commands:

```
$ swig -python -co python.swg
'python.swg' checked out from the SWIG library.
$ cat python.swg
```

Additional typemap examples can also be found in the `typemaps.i` file.

## 26.8.3 Typemap variables

Within typemap code, a number of special variables prefaced with a `$` may appear. A full list of variables can be found in the "[Typemaps](#)" chapter. This is a list of the most common variables:

`$1`

A C local variable corresponding to the actual type specified in the `%typemap` directive. For input values, this is a C local variable that's supposed to hold an argument value. For output values, this is the raw result that's supposed to be returned to Python.

`$input`

A `PyObject *` holding a raw Python object with an argument or variable value.

`$result`

A `PyObject *` that holds the result to be returned to Python.

`$1_name`

The parameter name that was matched.

`$1_type`

The actual C datatype matched by the `typemap`.

`$1_ltype`

An assignable version of the datatype matched by the `typemap` (a type that can appear on the left-hand-side of a C assignment operation). This type is stripped of qualifiers and may be an altered version of `$1_type`. All arguments and local variables in wrapper functions are declared using this type so that their values can be properly assigned.

`$symname`

The Python name of the wrapper function being created.

## 26.8.4 Useful Python Functions

When you write a `typemap`, you usually have to work directly with Python objects. The following functions may prove to be useful.

### Python Integer Functions

```
PyObject *PyInt_FromLong(long l);
long      PyInt_AsLong(PyObject *);
int       PyInt_Check(PyObject *);
```

### Python Floating Point Functions

```
PyObject *PyFloat_FromDouble(double);
double    PyFloat_AsDouble(PyObject *);
int       PyFloat_Check(PyObject *);
```

### Python String Functions

```
PyObject *PyString_FromString(char *);
PyObject *PyString_FromStringAndSize(char *, lint len);
int       PyString_Size(PyObject *);
char      *PyString_AsString(PyObject *);
int       PyString_Check(PyObject *);
```

### Python List Functions

```
PyObject *PyList_New(int size);
int       PyList_Size(PyObject *list);
PyObject *PyList_GetItem(PyObject *list, int i);
int       PyList_SetItem(PyObject *list, int i, PyObject *item);
int       PyList_Insert(PyObject *list, int i, PyObject *item);
int       PyList_Append(PyObject *list, PyObject *item);
PyObject *PyList_GetSlice(PyObject *list, int i, int j);
int       PyList_SetSlice(PyObject *list, int i, int , PyObject *list2);
int       PyList_Sort(PyObject *list);
int       PyList_Reverse(PyObject *list);
PyObject *PyList_AsTuple(PyObject *list);
int       PyList_Check(PyObject *);
```

## Python Tuple Functions

```
PyObject *PyTuple_New(int size);
int      PyTuple_Size(PyObject *);
PyObject *PyTuple_GetItem(PyObject *, int i);
int      PyTuple_SetItem(PyObject *, int i, PyObject *item);
PyObject *PyTuple_GetSlice(PyObject *t, int i, int j);
int      PyTuple_Check(PyObject *);
```

## Python Dictionary Functions

```
write me
```

## Python File Conversion Functions

```
PyObject *PyFile_FromFile(FILE *f);
FILE      *PyFile_AsFile(PyObject *);
int      PyFile_Check(PyObject *);
```

## Abstract Object Interface

```
write me
```

# 26.9 Typemap Examples

This section includes a few examples of typemaps. For more examples, you might look at the files "python.swg" and "typemaps.i" in the SWIG library.

### 26.9.1 Converting Python list to a char \*\*

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Python list object to be used as a char \*\* object.

```
%module argv

// This tells SWIG to treat char ** as a special case
%typemap(in) char ** {
    /* Check if is a list */
    if (PyList_Check($input)) {
        int size = PyList_Size($input);
        int i = 0;
        $1 = (char **) malloc((size+1)*sizeof(char *));
        for (i = 0; i < size; i++) {
            PyObject *o = PyList_GetItem($input,i);
            if (PyString_Check(o))
                $1[i] = PyString_AsString(PyList_GetItem($input,i));
            else {
                PyErr_SetString(PyExc_TypeError,"list must contain strings");
                free($1);
                return NULL;
            }
        }
        $1[i] = 0;
    } else {
        PyErr_SetString(PyExc_TypeError,"not a list");
        return NULL;
    }
}

// This cleans up the char ** array we malloc'd before the function call
%typemap(freearg) char ** {
    free((char *) $1);
}
```



```

}

// Now a test function
%inline %{
int print_args(char **argv) {
    int i = 0;
    while (argv[i]) {
        printf("argv[%d] = %s\n", i,argv[i]);
        i++;
    }
    return i;
}
%}

```

When this module is compiled, the wrapped C function now operates as follows :

```

>>> from argv import *
>>> print_args(["Dave", "Mike", "Mary", "Jane", "John"])
argv[0] = Dave
argv[1] = Mike
argv[2] = Mary
argv[3] = Jane
argv[4] = John
5

```

In the example, two different typemaps are used. The "in" typemap is used to receive an input argument and convert it to a C array. Since dynamic memory allocation is used to allocate memory for the array, the "freearg" typemap is used to later release this memory after the execution of the C function.

## 26.9.2 Expanding a Python object into multiple arguments

Suppose that you had a collection of C functions with arguments such as the following:

```
int foo(int argc, char **argv);
```

In the previous example, a typemap was written to pass a Python list as the `char **argv`. This allows the function to be used from Python as follows:

```
>>> foo(4, ["foo", "bar", "spam", "1"])
```

Although this works, it's a little awkward to specify the argument count. To fix this, a multi-argument typemap can be defined. This is not very difficult—you only have to make slight modifications to the previous example:

```

%typemap(in) (int argc, char **argv) {
    /* Check if is a list */
    if (PyList_Check($input)) {
        int i;
        $1 = PyList_Size($input);
        $2 = (char **) malloc(($1+1)*sizeof(char *));
        for (i = 0; i < $1; i++) {
            PyObject *o = PyList_GetItem($input,i);
            if (PyString_Check(o))
                $2[i] = PyString_AsString(PyList_GetItem($input,i));
            else {
                PyErr_SetString(PyExc_TypeError,"list must contain strings");
                free($2);
                return NULL;
            }
        }
        $2[i] = 0;
    } else {
        PyErr_SetString(PyExc_TypeError,"not a list");
    }
}

```

```

        return NULL;
    }
}

%typemap(freearg) (int argc, char **argv) {
    free((char *) $2);
}

```

When writing a multiple-argument typemap, each of the types is referenced by a variable such as \$1 or \$2. The typemap code simply fills in the appropriate values from the supplied Python object.

With the above typemap in place, you will find it no longer necessary to supply the argument count. This is automatically set by the typemap code. For example:

```
>>> foo(["foo", "bar", "spam", "1"])
```

### 26.9.3 Using typemaps to return arguments

A common problem in some C programs is that values may be returned in arguments rather than in the return value of a function. For example:

```

/* Returns a status value and two values in out1 and out2 */
int spam(double a, double b, double *out1, double *out2) {
    ... Do a bunch of stuff ...
    *out1 = result1;
    *out2 = result2;
    return status;
};

```

A typemap can be used to handle this case as follows :

```

%module outarg

// This tells SWIG to treat an double * argument with name 'OutValue' as
// an output value. We'll append the value to the current result which
// is guaranteed to be a List object by SWIG.

%typemap(argout) double *OutValue {
    PyObject *o, *o2, *o3;
    o = PyFloat_FromDouble(*$1);
    if ((!$result) || ($result == Py_None)) {
        $result = o;
    } else {
        if (!PyTuple_Check($result)) {
            PyObject *o2 = $result;
            $result = PyTuple_New(1);
            PyTuple_SetItem(target,0,o2);
        }
        o3 = PyTuple_New(1);
        PyTuple_SetItem(o3,0,o);
        o2 = $result;
        $result = PySequence_Concat(o2,o3);
        Py_DECREF(o2);
        Py_DECREF(o3);
    }
}

int spam(double a, double b, double *OutValue, double *OutValue);

```

The typemap works as follows. First, a check is made to see if any previous result exists. If so, it is turned into a tuple and the new output value is concatenated to it. Otherwise, the result is returned normally. For the sample function `spam()`, there are three

output values—meaning that the function will return a 3-tuple of the results.

As written, the function must accept 4 arguments as input values, last two being pointers to doubles. If these arguments are only used to hold output values (and have no meaningful input value), an additional `typemap` can be written. For example:

```
%typemap(in,numinputs=0) double *OutValue(double temp) {
    $1 = &temp;
}
```

By specifying `numinputs=0`, the input value is ignored. However, since the argument still has to be set to some meaningful value before calling C, it is set to point to a local variable `temp`. When the function stores its output value, it will simply be placed in this local variable. As a result, the function can now be used as follows:

```
>>> a = spam(4,5)
>>> print a
(0, 2.45, 5.0)
>>> x,y,z = spam(4,5)
>>>
```

## 26.9.4 Mapping Python tuples into small arrays

In some applications, it is sometimes desirable to pass small arrays of numbers as arguments. For example :

```
extern void set_direction(double a[4]);          // Set direction vector
```

This too, can be handled used `typemaps` as follows :

```
// Grab a 4 element array as a Python 4-tuple
%typemap(in) double[4](double temp[4]) {    // temp[4] becomes a local variable
    int i;
    if (PyTuple_Check($input)) {
        if (!PyArg_ParseTuple($input,"dddd",temp,temp+1,temp+2,temp+3)) {
            PyErr_SetString(PyExc_TypeError,"tuple must have 4 elements");
            return NULL;
        }
        $1 = &temp[0];
    } else {
        PyErr_SetString(PyExc_TypeError,"expected a tuple.");
        return NULL;
    }
}
```

This allows our `set_direction` function to be called from Python as follows :

```
>>> set_direction((0.5,0.0,1.0,-0.25))
```

Since our mapping copies the contents of a Python tuple into a C array, such an approach would not be recommended for huge arrays, but for small structures, this approach works fine.

## 26.9.5 Mapping sequences to C arrays

Suppose that you wanted to generalize the previous example to handle C arrays of different sizes. To do this, you might write a `typemap` as follows:

```
// Map a Python sequence into any sized C double array
%typemap(in) double[ANY](double temp[$1_dim0]) {
    int i;
    if (!PySequence_Check($input)) {
        PyErr_SetString(PyExc_TypeError,"Expecting a sequence");
    }
}
```

```

    return NULL;
}
if (PyObject_Length($input) != $1_dim0) {
    PyErr_SetString(PyExc_ValueError, "Expecting a sequence with $1_dim0 elements");
    return NULL;
}
for (i = 0; i < $1_dim0; i++) {
    PyObject *o = PySequence_GetItem($input, i);
    if (!PyFloat_Check(o)) {
        PyErr_SetString(PyExc_ValueError, "Expecting a sequence of floats");
        return NULL;
    }
    temp[i] = PyFloat_AsDouble(o);
}
$1 = &temp[0];
}

```

In this case, the variable `$1_dim0` is expanded to match the array dimensions actually used in the C code. This allows the `typemap` to be applied to types such as:

```

void foo(double x[10]);
void bar(double a[4], double b[8]);

```

Since the above `typemap` code gets inserted into every wrapper function where used, it might make sense to use a helper function instead. This will greatly reduce the amount of wrapper code. For example:

```

%{
static int convert_darray(PyObject *input, double *ptr, int size) {
    int i;
    if (!PySequence_Check(input)) {
        PyErr_SetString(PyExc_TypeError, "Expecting a sequence");
        return 0;
    }
    if (PyObject_Length(input) != size) {
        PyErr_SetString(PyExc_ValueError, "Sequence size mismatch");
        return 0;
    }
    for (i = 0; i < size; i++) {
        PyObject *o = PySequence_GetItem(input, i);
        if (!PyFloat_Check(o)) {
            PyErr_SetString(PyExc_ValueError, "Expecting a sequence of floats");
            return 0;
        }
        ptr[i] = PyFloat_AsDouble(o);
    }
    return 1;
}
}%

%typemap(in) double [ANY](double temp[$1_dim0]) {
    if (!convert_darray($input, temp, $1_dim0)) {
        return NULL;
    }
    $1 = &temp[0];
}

```

## 26.9.6 Pointer handling

Occasionally, it might be necessary to convert pointer values that have been stored using the SWIG typed-pointer representation. Since there are several ways in which pointers can be represented, the following two functions are used to safely perform this conversion:

```

int SWIG_ConvertPtr(PyObject *obj, void **ptr, swig_type_info *ty, int flags)

```

Converts a Python object `obj` to a C pointer. The result of the conversion is placed into the pointer located at `ptr`. `ty` is a SWIG type descriptor structure. `flags` is used to handle error checking and other aspects of conversion. It is the bitwise-or of several flag values including `SWIG_POINTER_EXCEPTION` and `SWIG_POINTER_DISOWN`. The first flag makes the function raise an exception on type error. The second flag additionally steals ownership of an object. Returns 0 on success and -1 on error.

```
PyObject *Swig_NewPointerObj(void *ptr, swig_type_info *ty, int own)
```

Creates a new Python pointer object. `ptr` is the pointer to convert, `ty` is the SWIG type descriptor structure that describes the type, and `own` is a flag that indicates whether or not Python should take ownership of the pointer.

Both of these functions require the use of a special SWIG type-descriptor structure. This structure contains information about the mangled name of the datatype, type-equivalence information, as well as information about converting pointer values under C++ inheritance. For a type of `Foo *`, the type descriptor structure is usually accessed as follows:

```
Foo *f;
if (SWIG_ConvertPtr($input, (void **) &f, SWIGTYPE_p_Foo, SWIG_POINTER_EXCEPTION) == -1)
    return NULL;

PyObject *obj;
obj = SWIG_NewPointerObj(f, SWIGTYPE_p_Foo, 0);
```

In a `typemap`, the type descriptor should always be accessed using the special `typemap` variable `$l_descriptor`. For example:

```
%typemap(in) Foo * {
    if ((SWIG_ConvertPtr($input, (void **) &$l, $l_descriptor, SWIG_POINTER_EXCEPTION)) == -1)
        return NULL;
}
```

If necessary, the descriptor for any type can be obtained using the `$descriptor()` macro in a `typemap`. For example:

```
%typemap(in) Foo * {
    if ((SWIG_ConvertPtr($input, (void **) &$l, $descriptor(Foo *),
                        SWIG_POINTER_EXCEPTION)) == -1)
        return NULL;
}
```

Although the pointer handling functions are primarily intended for manipulating low-level pointers, both functions are fully aware of Python proxy classes. Specifically, `SWIG_ConvertPtr()` will retrieve a pointer from any object that has a `this` attribute. In addition, `SWIG_NewPointerObj()` can automatically generate a proxy class object (if applicable).

## 26.10 Docstring Features

Using docstrings in Python code is becoming more and more important and more tools are coming on the scene that take advantage of them, everything from full-blown documentation generators to class browsers and popup call-tips in Python-aware IDEs. Given the way that SWIG generates the proxy code by default, your users will normally get something like `"function_name(*args)"` in the popup calltip of their IDE which is next to useless when the real function prototype might be something like this:

```
bool function_name(int x, int y, Foo* foo=NULL, Bar* bar=NULL);
```

The features described in this section make it easy for you to add docstrings to your modules, functions and methods that can then be used by the various tools out there to make the programming experience of your users much simpler.

## 26.10.1 Module docstring

Python allows a docstring at the beginning of the `.py` file before any other statements, and it is typically used to give a general description of the entire module. SWIG supports this by setting an option of the `%module` directive. For example:

```
%module(docstring="This is the example module's docstring") example
```

When you have more than just a line or so then you can retain the easy readability of the `%module` directive by using a macro. For example:

```
%define DOCSTRING
"The `XmlResource` class allows program resources defining menus,
layout of controls on a panel, etc. to be loaded from an XML file."
%enddef

%module(docstring=DOCSTRING) xrc
```

## 26.10.2 %feature("autodoc")

As alluded to above SWIG will generate all the function and method proxy wrappers with just `"*args"` (or `"*args, **kwargs"` if the `-keyword` option is used) for a parameter list and will then sort out the individual parameters in the C wrapper code. This is nice and simple for the wrapper code, but makes it difficult to be programmer and tool friendly as anyone looking at the `.py` file will not be able to find out anything about the parameters that the functions accept.

But since SWIG does know everything about the function it is possible to generate a docstring containing the parameter types, names and default values. Since many of the docstring tools are adopting a standard of recognizing if the first thing in the docstring is a function prototype then using that instead of what they found from introspection, then life is good once more.

SWIG's Python module provides support for the "autodoc" feature, which when attached to a node in the parse tree will cause a docstring to be generated that includes the name of the function, parameter names, default values if any, and return type if any. There are also three options for autodoc controlled by the value given to the feature, described below.

### 26.10.2.1 %feature("autodoc", "0")

When the "0" option is given then the types of the parameters will *not* be included in the autodoc string. For example, given this function prototype:

```
%feature("autodoc", "0");
bool function_name(int x, int y, Foo* foo=NULL, Bar* bar=NULL);
```

Then Python code like this will be generated:

```
def function_name(*args, **kwargs):
    """function_name(x, y, foo=None, bar=None) -> bool"""
    ...
```

### 26.10.2.2 %feature("autodoc", "1")

When the "1" option is used then the parameter types *will* be used in the autodoc string. In addition, an attempt is made to simplify the type name such that it makes more sense to the Python user. Pointer, reference and const info is removed, `%rename`'s are evaluated, etc. (This is not always successful, but works most of the time. See the next section for what to do when it doesn't.) Given the example above, then turning on the parameter types with the "1" option will result in Python code like this:

```
def function_name(*args, **kwargs):
    """function_name(int x, int y, Foo foo=None, Bar bar=None) -> bool"""
    ...
```

### 26.10.2.3 %feature("autodoc", "docstring")

Finally, there are times when the automatically generated autodoc string will make no sense for a Python programmer, particularly when a typemap is involved. So if you give an explicit value for the autodoc feature then that string will be used in place of the automatically generated string. For example:

```
%feature("autodoc", "GetPosition() -> (x, y)") GetPosition;
void GetPosition(int* OUTPUT, int* OUTPUT);
```

### 26.10.3 %feature("docstring")

In addition to the autodoc strings described above, you can also attach any arbitrary descriptive text to a node in the parse tree with the "docstring" feature. When the proxy module is generated then any docstring associated with classes, function or methods are output. If an item already has an autodoc string then it is combined with the docstring and they are output together. If the docstring is all on a single line then it is output like this::

```
"""This is the docstring"""
```

Otherwise, to aid readability it is output like this:

```
"""
This is a multi-line docstring
with more than one line.
"""
```

## 26.11 Python Packages

Using the package option of the %module directive allows you to specify what Python package that the module will be living in when installed.

```
%module(package="wx") xrc
```

This is useful when the .i file is %imported by another .i file. By default SWIG will assume that the importer is able to find the importee with just the module name, but if they live in separate Python packages then that won't work. However if the importee specifies what its package is with the %module option then the Python code generated for the importer will use that package name when importing the other module and also in base class declarations, etc. if the package name is different than its own.

## 27 SWIG and Ruby

- [Preliminaries](#)
  - ◆ [Running SWIG](#)
  - ◆ [Getting the right header files](#)
  - ◆ [Compiling a dynamic module](#)
  - ◆ [Using your module](#)
  - ◆ [Static linking](#)
  - ◆ [Compilation of C++ extensions](#)
- [Building Ruby Extensions under Windows 95/NT](#)
  - ◆ [Running SWIG from Developer Studio](#)
- [The Ruby-to-C/C++ Mapping](#)
  - ◆ [Modules](#)
  - ◆ [Functions](#)
  - ◆ [Variable Linking](#)
  - ◆ [Constants](#)
  - ◆ [Pointers](#)
  - ◆ [Structures](#)
  - ◆ [C++ classes](#)
  - ◆ [C++ Inheritance](#)
  - ◆ [C++ Overloaded Functions](#)
  - ◆ [C++ Operators](#)
  - ◆ [C++ namespaces](#)
  - ◆ [C++ templates](#)
  - ◆ [C++ Smart Pointers](#)
  - ◆ [Cross-Language Polymorphism](#)
    - ◇ [Exception Unrolling](#)
- [Input and output parameters](#)
- [Simple exception handling](#)
- [Typemaps](#)
  - ◆ [What is a typemap?](#)
  - ◆ [Ruby typemaps](#)
  - ◆ [Typemap variables](#)
  - ◆ [Useful Functions](#)
    - ◇ [C Datatypes to Ruby Objects](#)
    - ◇ [Ruby Objects to C Datatypes](#)
    - ◇ [Macros for VALUE](#)
    - ◇ [Exceptions](#)
    - ◇ [Iterators](#)
  - ◆ [Typemap Examples](#)
  - ◆ [Converting a Ruby array to a char \\*\\*](#)
  - ◆ [Collecting arguments in a hash](#)
  - ◆ [Pointer handling](#)
    - ◇ [Ruby Datatype Wrapping](#)
- [Operator overloading](#)
  - ◆ [Example: STL Vector to Ruby Array](#)
- [Advanced Topics](#)
  - ◆ [Creating Multi-Module Packages](#)
  - ◆ [Defining Aliases](#)
  - ◆ [Predicate Methods](#)
  - ◆ [Specifying Mixin Modules](#)
  - ◆ [Interacting with Ruby's Garbage Collector](#)

This chapter describes SWIG's support of Ruby.



## 27.1 Preliminaries

SWIG 1.3 is known to work with Ruby versions 1.6 and later. Given the choice, you should use the latest stable version of Ruby. You should also determine if your system supports shared libraries and dynamic loading. SWIG will work with or without dynamic loading, but the compilation process will vary.

This chapter covers most SWIG features, but in less depth than is found in earlier chapters. At the very least, make sure you also read the "[SWIG Basics](#)" chapter. It is also assumed that the reader has a basic understanding of Ruby.

### 27.1.1 Running SWIG

To build a Ruby module, run SWIG using the `-ruby` option:

```
$ swig -ruby example.i
```

If building a C++ extension, add the `-c++` option:

```
$ swig -c++ -ruby example.i
```

This creates a file `example_wrap.c` (`example_wrap.cxx` if compiling a C++ extension) that contains all of the code needed to build a Ruby extension module. To finish building the module, you need to compile this file and link it with the rest of your program.

### 27.1.2 Getting the right header files

In order to compile the wrapper code, the compiler needs the `ruby.h` header file. This file is usually contained in a directory such as

```
/usr/local/lib/ruby/1.6/i686-linux/ruby.h
```

The exact location may vary on your machine, but the above location is typical. If you are not entirely sure where Ruby is installed, you can run Ruby to find out. For example:

```
$ ruby -e 'puts $:.join("\n")'
/usr/local/lib/ruby/site_ruby/1.6
/usr/local/lib/ruby/site_ruby/1.6/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/1.6
/usr/local/lib/ruby/1.6/i686-linux
.
```

### 27.1.3 Compiling a dynamic module

Ruby extension modules are typically compiled into shared libraries that the interpreter loads dynamically at runtime. Since the exact commands for doing this vary from platform to platform, your best bet is to follow the steps described in the `README.EXT` file from the Ruby distribution:

1. Create a file called `extconf.rb` that looks like the following:

```
require 'mkmf'
create_makefile('example')
```

2. Type the following to build the extension:

```
$ ruby extconf.rb
```

```
$ make
$ make install
```

Of course, there is the problem that `mkmf` does not work correctly on all platforms, e.g. HP-UX. If you need to add your own make rules to the file that `extconf.rb` produces, you can add this:

```
open("Makefile", "a") { |mf|
  puts <<EOM
  # Your make rules go here
  EOM
}
```

to the end of the `extconf.rb` file. If for some reason you don't want to use the standard approach, you'll need to determine the correct compiler and linker flags for your build platform. For example, a typical sequence of commands for the Linux operating system would look something like this:

```
$ swig -ruby example.i
$ gcc -c example.c
$ gcc -c example_wrap.c -I/usr/local/lib/ruby/1.6/i686-linux
$ gcc -shared example.o example_wrap.o -o example.so
```

For other platforms it may be necessary to compile with the `-fPIC` option to generate position-independent code. If in doubt, consult the manual pages for your compiler and linker to determine the correct set of options. You might also check the [SWIG Wiki](#) for additional information.

## 27.1.4 Using your module

Ruby *module* names must be capitalized, but the convention for Ruby *feature* names is to use lowercase names. So, for example, the **Etc** extension module is imported by requiring the **etc** feature:

```
# The feature name begins with a lowercase letter...
require 'etc'

# ... but the module name begins with an uppercase letter
puts "Your login name: #{Etc.getlogin}"
```

To stay consistent with this practice, you should always specify a **lowercase** module name with SWIG's `%module` directive. SWIG will automatically correct the resulting Ruby module name for your extension. So for example, a SWIG interface file that begins with:

```
%module example
```

will result in an extension module using the feature name "example" and Ruby module name "Example".

## 27.1.5 Static linking

An alternative approach to dynamic linking is to rebuild the Ruby interpreter with your extension module added to it. In the past, this approach was sometimes necessary due to limitations in dynamic loading support on certain machines. However, the situation has improved greatly over the last few years and you should not consider this approach unless there is really no other option.

The usual procedure for adding a new module to Ruby involves finding the Ruby source, adding an entry to the `ext/Setup` file, adding your directory to the list of extensions in the file, and finally rebuilding Ruby.

## 27.1.6 Compilation of C++ extensions

On most machines, C++ extension modules should be linked using the C++ compiler. For example:

```
$ swig -c++ -ruby example.i
$ g++ -c example.cxx
$ g++ -c example_wrap.cxx -I/usr/local/lib/ruby/1.6/i686-linux
$ g++ -shared example.o example_wrap.o -o example.so
```

If you've written an `extconf.rb` script to automatically generate a `Makefile` for your C++ extension module, keep in mind that (as of this writing) Ruby still uses `gcc` and not `g++` as its linker. As a result, the required C++ runtime library support will not be automatically linked into your extension module and it may fail to load on some platforms. A workaround for this problem is use the `mkmf` module's `append_library()` method to add one of the C++ runtime libraries to the list of libraries linked into your extension, e.g.

```
require 'mkmf'
$libs = append_library($libs, "supc++")
create_makefile('example')
```

## 27.2 Building Ruby Extensions under Windows 95/NT

Building a SWIG extension to Ruby under Windows 95/NT is roughly similar to the process used with Unix. Normally, you will want to produce a DLL that can be loaded into the Ruby interpreter. For all recent versions of Ruby, the procedure described above (i.e. using an `extconf.rb` script) will work with Windows as well; you should be able to build your code into a DLL by typing:

```
C:\swigtest> ruby extconf.rb
C:\swigtest> nmake
C:\swigtest> nmake install
```

The remainder of this section covers the process of compiling SWIG-generated Ruby extensions with Microsoft Visual C++ 6 (i.e. within the Developer Studio IDE, instead of using the command line tools). In order to build extensions, you may need to download the source distribution to the Ruby package, as you will need the Ruby header files.

### 27.2.1 Running SWIG from Developer Studio

If you are developing your application within Microsoft developer studio, SWIG can be invoked as a custom build option. The process roughly follows these steps :

- Open up a new workspace and use the AppWizard to select a DLL project.
- Add both the SWIG interface file (the `.i` file), any supporting C files, and the name of the wrapper file that will be created by SWIG (i.e.. `example_wrap.c`). Note : If using C++, choose a different suffix for the wrapper file such as `example_wrap.cxx`. Don't worry if the wrapper file doesn't exist yet—Developer Studio will keep a reference to it around.
- Select the SWIG interface file and go to the settings menu. Under settings, select the "Custom Build" option.
- Enter "SWIG" in the description field.
- Enter "`swig -ruby -o $(ProjDir)\$(InputName)_wrap.c $(InputPath)`" in the "Build command(s) field". You may have to include the path to `swig.exe`.
- Enter "`$(ProjDir)\$(InputName)_wrap.c`" in the "Output files(s) field".
- Next, select the settings for the entire project and go to the C/C++ tab and select the Preprocessor category. Add `NT=1` to the Preprocessor definitions. This must be set else you will get compilation errors. Also add `IMPORT` to the preprocessor definitions, else you may get runtime errors. Also add the include directories for your Ruby installation under "Additional include directories".
- Next, select the settings for the entire project and go to the Link tab and select the General category. Set the name of the

output file to match the name of your Ruby module (i.e.. example.dll). Next add the Ruby library file to your link libraries under Object/Library modules. For example "mswin32-ruby16.lib. You also need to add the path to the library under the Input tab – Additional library path.

- Build your project.

Now, assuming all went well, SWIG will be automatically invoked when you build your project. Any changes made to the interface file will result in SWIG being automatically invoked to produce a new version of the wrapper file. To run your new Ruby extension, simply run Ruby and use the `require` command as normal. For example if you have this ruby file `run.rb`:

```
# file: run.rb
require 'Example'

# Call a c function
print "Foo = ", Example.Foo, "\n"
```

Ensure the dll just built is in your path or current directory, then run the Ruby script from the DOS/Command prompt:

```
C:\swigtest> ruby run.rb
Foo = 3.0
```

## 27.3 The Ruby-to-C/C++ Mapping

This section describes the basics of how SWIG maps C or C++ declarations in your SWIG interface files to Ruby constructs.

### 27.3.1 Modules

The SWIG `%module` directive specifies the name of the Ruby module. If you specify:

```
%module example
```

then everything is wrapped into a Ruby module named `Example` that is nested directly under the global module. You can specify a more deeply nested module by specifying the fully-qualified module name in quotes, e.g.

```
%module "foo::bar::spam"
```

An alternate method of specifying a nested module name is to use the `-prefix` option on the SWIG command line. The prefix that you specify with this option will be prepended to the module name specified with the `%module` directive in your SWIG interface file. So for example, this declaration at the top of your SWIG interface file:

```
%module "foo::bar::spam"
```

will result in a nested module name of `Foo::Bar::Spam`, but you can achieve the same effect by specifying:

```
%module spam
```

and then running SWIG with the `-prefix` command line option:

```
$ swig -ruby -prefix "foo::bar::" example.i
```

Starting with SWIG 1.3.20, you can also choose to wrap everything into the global module by specifying the `-globalmodule` option on the SWIG command line, i.e.

```
$ swig -ruby -globalmodule example.i
```

Note that this does not relieve you of the requirement of specifying the SWIG module name with the `%module` directive (or the `-module` command-line option) as described earlier.

When choosing a module name, do not use the same name as a built-in Ruby command or standard module name, as the results may be unpredictable. Similarly, if you're using the `-globalmodule` option to wrap everything into the global module, take care that the names of your constants, classes and methods don't conflict with any of Ruby's built-in names.

## 27.3.2 Functions

Global functions are wrapped as Ruby module methods. For example, given the SWIG interface file `example.i`:

```
%module example

int fact(int n);
```

and C source file `example.c`:

```
int fact(int n) {
    if (n == 0)
        return 1;
    return (n * fact(n-1));
}
```

SWIG will generate a method *fact* in the *Example* module that can be used like so:

```
$ irb
irb(main):001:0> require 'example'
true
irb(main):002:0> Example.fact(4)
24
```

## 27.3.3 Variable Linking

C/C++ global variables are wrapped as a pair of singleton methods for the module: one to get the value of the global variable and one to set it. For example, the following SWIG interface file declares two global variables:

```
// SWIG interface file with global variables
%module example
...
extern int    variable1;
extern double Variable2;
...
```

Now look at the Ruby interface:

```
$ irb
irb(main):001:0> require 'Example'
true
irb(main):002:0> Example.variable1 = 2
2
irb(main):003:0> Example.Variable2 = 4 * 10.3
41.2
irb(main):004:0> Example.Variable2
41.2
```

If you make an error in variable assignment, you will receive an error message. For example:

```
irb(main):005:0> Example.Variable2 = "hello"
TypeError: no implicit conversion to float from string
    from (irb):5:in `Variable2='
    from (irb):5
```

If a variable is declared as `const`, it is wrapped as a read-only variable. Attempts to modify its value will result in an error.

To make ordinary variables read-only, you can also use the `%immutable` directive. For example:

```
%immutable;
extern char *path;
%mutable;
```

The `%immutable` directive stays in effect until it is explicitly disabled using `%mutable`.

### 27.3.4 Constants

C/C++ constants are wrapped as module constants initialized to the appropriate value. To create a constant, use `#define` or the `%constant` directive. For example:

```
#define PI 3.14159
#define VERSION "1.0"

%constant int FOO = 42;
%constant const char *path = "/usr/local";

const int BAR = 32;
```

Remember to use the `::` operator in Ruby to get at these constant values, e.g.

```
$ irb
irb(main):001:0> require 'Example'
true
irb(main):002:0> Example::PI
3.14159
```

### 27.3.5 Pointers

"Opaque" pointers to arbitrary C/C++ types (i.e. types that aren't explicitly declared in your SWIG interface file) are wrapped as data objects. So, for example, consider a SWIG interface file containing only the declarations:

```
Foo *get_foo();
void set_foo(Foo *foo);
```

For this case, the `get_foo()` method returns an instance of an internally generated Ruby class:

```
irb(main):001:0> foo = Example::get_foo()
#<SWIG::TYPE_p_Foo:0x402b1654>
```

A NULL pointer is always represented by the Ruby `nil` object.

### 27.3.6 Structures

C/C++ structs are wrapped as Ruby classes, with accessor methods (i.e. "getters" and "setters") for all of the struct members. For example, this struct declaration:

```
struct Vector {
    double x, y;
};
```

gets wrapped as a `Vector` class, with Ruby instance methods `x`, `x=`, `y` and `y=`. These methods can be used to access structure data from Ruby as follows:

```
$ irb
irb(main):001:0> require 'Example'
true
irb(main):002:0> f = Example::Vector.new
#<Example::Vector:0x4020b268>
irb(main):003:0> f.x = 10
nil
irb(main):004:0> f.x
10.0
```

Similar access is provided for unions and the public data members of C++ classes.

`const` members of a structure are read-only. Data members can also be forced to be read-only using the `%immutable` directive (in C++, `private` may also be used). For example:

```
struct Foo {
    ...
    %immutable;
    int x;          /* Read-only members */
    char *name;
    %mutable;
    ...
};
```

When `char *` members of a structure are wrapped, the contents are assumed to be dynamically allocated using `malloc` or `new` (depending on whether or not SWIG is run with the `-c++` option). When the structure member is set, the old contents will be released and a new value created. If this is not the behavior you want, you will have to use a `typemap` (described shortly).

Array members are normally wrapped as read-only. For example, this code:

```
struct Foo {
    int x[50];
};
```

produces a single accessor function like this:

```
int *Foo_x_get(Foo *self) {
    return self->x;
};
```

If you want to set an array member, you will need to supply a "memberin" `typemap` described in the [section on typemaps](#). As a special case, SWIG does generate code to set array members of type `char` (allowing you to store a Ruby string in the structure).

When structure members are wrapped, they are handled as pointers. For example,

```
struct Foo {
    ...
};

struct Bar {
    Foo f;
};
```

generates accessor functions such as this:

```
Foo *Bar_f_get(Bar *b) {
    return &b->f;
}

void Bar_f_set(Bar *b, Foo *val) {
    b->f = *val;
}
```

## 27.3.7 C++ classes

Like structs, C++ classes are wrapped by creating a new Ruby class of the same name with accessor methods for the public class member data. Additionally, public member functions for the class are wrapped as Ruby instance methods, and public static member functions are wrapped as Ruby singleton methods. So, given the C++ class declaration:

```
class List {
public:
    List();
    ~List();
    int  search(char *item);
    void insert(char *item);
    void remove(char *item);
    char *get(int n);
    int  length;
    static void print(List *l);
};
```

SWIG would create a `List` class with:

- instance methods *search*, *insert*, *remove*, and *get*;
- instance methods *length* and *length=* (to get and set the value of the *length* data member); and,
- a *print* singleton method for the class.

In Ruby, these functions are used as follows:

```
require 'Example'

l = Example::List.new

l.insert("Ale")
l.insert("Stout")
l.insert("Lager")
Example.print(l)
l.length()
----- produces the following output
Lager
Stout
Ale
3
```

## 27.3.8 C++ Inheritance

The SWIG type-checker is fully aware of C++ inheritance. Therefore, if you have classes like this:

```
class Parent {
    ...
};

class Child : public Parent {
    ...
};
```

those classes are wrapped into a hierarchy of Ruby classes that reflect the same inheritance structure. All of the usual Ruby utility methods work normally:

```
irb(main):001:0> c = Child.new
#<Bar:0x4016efd4>
irb(main):002:0> c.instance_of? Child
true
irb(main):003:0> b.instance_of? Parent
false
```



```

irb(main):004:0> b.is_a? Child
true
irb(main):005:0> b.is_a? Parent
true
irb(main):006:0> Child < Parent
true
irb(main):007:0> Child > Parent
false

```

Furthermore, if you have a function like this:

```
void spam(Parent *f);
```

then the function `spam( )` accepts `Parent*` or a pointer to any class derived from `Parent`.

Until recently, the Ruby module for SWIG didn't support multiple inheritance, and this is still the default behavior. This doesn't mean that you can't wrap C++ classes which inherit from multiple base classes; it simply means that only the **first** base class listed in the class declaration is considered, and any additional base classes are ignored. As an example, consider a SWIG interface file with a declaration like this:

```

class Derived : public Base1, public Base2
{
    ...
};

```

For this case, the resulting Ruby class (`Derived`) will only consider `Base1` as its superclass. It won't inherit any of `Base2`'s member functions or data and it won't recognize `Base2` as an "ancestor" of `Derived` (i.e. the `is_a?` relationship would fail). When SWIG processes this interface file, you'll see a warning message like:

```

example.i:5: Warning(802): Warning for Derived: Base Base2 ignored.
Multiple inheritance is not supported in Ruby.

```

Starting with SWIG 1.3.20, the Ruby module for SWIG provides limited support for multiple inheritance. Because the approach for dealing with multiple inheritance introduces some limitations, this is an optional feature that you can activate with the `-minherit` command-line option:

```
$ swig -c++ -ruby -minherit example.i
```

Using our previous example, if your SWIG interface file contains a declaration like this:

```

class Derived : public Base1, public Base2
{
    ...
};

```

and you run SWIG with the `-minherit` command-line option, then you will end up with a Ruby class `Derived` that appears to "inherit" the member data and functions from both `Base1` and `Base2`. What actually happens is that three different top-level classes are created, with Ruby's `Object` class as their superclass. Each of these classes defines a nested module named `Impl`, and it's in these nested `Impl` modules that the actual instance methods for the classes are defined, i.e.

```

class Base1
  module Impl
    # Define Base1 methods here
  end
  include Impl
end

class Base2
  module Impl
    # Define Base2 methods here
  end
end

```

```

    include Impl
end

class Derived
  module Impl
    include Base1::Impl
    include Base2::Impl
    # Define Derived methods here
  end
  include Impl
end

```

Observe that after the nested `Impl` module for a class is defined, it is mixed-in to the class itself. Also observe that the `Derived::Impl` module first mixes-in its base classes' `Impl` modules, thus "inheriting" all of their behavior.

The primary drawback is that, unlike the default mode of operation, neither `Base1` nor `Base2` is a true superclass of `Derived` anymore:

```

obj = Derived.new
obj.is_a? Base1    # this will return false...
obj.is_a? Base2    # ... and so will this

```

In most cases, this is not a serious problem since objects of type `Derived` will otherwise behave as though they inherit from both `Base1` and `Base2` (i.e. they exhibit ["Duck Typing"](#)).

### 27.3.9 C++ Overloaded Functions

C++ overloaded functions, methods, and constructors are mostly supported by SWIG. For example, if you have two functions like this:

```

void foo(int);
void foo(char *c);

```

You can use them in Ruby in a straightforward manner:

```

irb(main):001:0> foo(3)           # foo(int)
irb(main):002:0> foo("Hello")    # foo(char *c)

```

Similarly, if you have a class like this,

```

class Foo {
public:
    Foo();
    Foo(const Foo &);
    ...
};

```

you can write Ruby code like this:

```

irb(main):001:0> f = Foo.new      # Create a Foo
irb(main):002:0> g = Foo.new(f)  # Copy f

```

Overloading support is not quite as flexible as in C++. Sometimes there are methods that SWIG can't disambiguate. For example:

```

void spam(int);
void spam(short);

```

or

```

void foo(Bar *b);
void foo(Bar &b);

```

If declarations such as these appear, you will get a warning message like this:

```
example.i:12: Warning(509): Overloaded spam(short) is shadowed by spam(int)
at example.i:11.
```

To fix this, you either need to ignore or rename one of the methods. For example:

```
%rename(spam_short) spam(short);
...
void spam(int);
void spam(short); // Accessed as spam_short
```

or

```
%ignore spam(short);
...
void spam(int);
void spam(short); // Ignored
```

SWIG resolves overloaded functions and methods using a disambiguation scheme that ranks and sorts declarations according to a set of type-precedence rules. The order in which declarations appear in the input does not matter except in situations where ambiguity arises—in this case, the first declaration takes precedence.

Please refer to the ["SWIG and C++"](#) chapter for more information about overloading.

### 27.3.10 C++ Operators

For the most part, overloaded operators are handled automatically by SWIG and do not require any special treatment on your part. So if your class declares an overloaded addition operator, e.g.

```
class Complex {
    ...
    Complex operator+(Complex &);
    ...
};
```

the resulting Ruby class will also support the addition (+) method correctly.

For cases where SWIG's built-in support is not sufficient, C++ operators can be wrapped using the `%rename` directive (available on SWIG 1.3.10 and later releases). All you need to do is give the operator the name of a valid Ruby identifier. For example:

```
%rename(add_complex) operator+(Complex &, Complex &);
...
Complex operator+(Complex &, Complex &);
```

Now, in Ruby, you can do this:

```
a = Example::Complex.new(2, 3)
b = Example::Complex.new(4, -1)
c = Example.add_complex(a, b)
```

More details about wrapping C++ operators into Ruby operators is discussed in the [section on operator overloading](#).

### 27.3.11 C++ namespaces

SWIG is aware of C++ namespaces, but namespace names do not appear in the module nor do namespaces result in a module that is broken up into submodules or packages. For example, if you have a file like this,

```
%module example
```

```
namespace foo {
    int fact(int n);
    struct Vector {
        double x,y,z;
    };
};
```

it works in Ruby as follows:

```
irb(main):001:0> require 'example'
true
irb(main):002:0> Example.fact(3)
6
irb(main):003:0> v = Example::Vector.new
#<Example::Vector:0x4016f4d4>
irb(main):004:0> v.x = 3.4
3.4
irb(main):004:0> v.y
0.0
```

If your program has more than one namespace, name conflicts (if any) can be resolved using `%rename` For example:

```
%rename(Bar_spam) Bar::spam;

namespace Foo {
    int spam();
}

namespace Bar {
    int spam();
}
```

If you have more than one namespace and you want to keep their symbols separate, consider wrapping them as separate SWIG modules. For example, make the module name the same as the namespace and create extension modules for each namespace separately. If your program utilizes thousands of small deeply nested namespaces each with identical symbol names, well, then you get what you deserve.

### 27.3.12 C++ templates

C++ templates don't present a huge problem for SWIG. However, in order to create wrappers, you have to tell SWIG to create wrappers for a particular template instantiation. To do this, you use the `%template` directive. For example:

```
%module example

%{
#include "pair.h"
%}

template<class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1&, const T2&);
    ~pair();
};

%template(Pairii) pair<int,int>;
```

In Ruby:

```
irb(main):001:0> require 'example'
true
irb(main):002:0> p = Example::Pairii.new(3, 4)
#<Example::Pairii:0x4016f4df>
irb(main):003:0> p.first
3
irb(main):004:0> p.second
4
```

On a related note, the standard SWIG library contains a number of modules that provide typemaps for standard C++ library classes (such as `std::pair`, `std::string` and `std::vector`). These library modules don't provide wrappers around the templates themselves, but they do make it convenient for users of your extension module to pass Ruby objects (such as arrays and strings) to wrapped C++ code that expects instances of standard C++ templates. For example, suppose the C++ library you're wrapping has a function that expects a vector of floats:

```
%module example

float sum(const std::vector<float>& values);
```

Rather than go through the hassle of writing an "in" typemap to convert an array of Ruby numbers into a `std::vector<float>`, you can just use the `std_vector.i` module from the standard SWIG library:

```
%module example

#include std_vector.i

float sum(const std::vector<float>& values);
```

Obviously, there is a lot more to template wrapping than shown in these examples. More details can be found in the [SWIG and C++](#) chapter.

### 27.3.13 C++ Smart Pointers

In certain C++ programs, it is common to use classes that have been wrapped by so-called "smart pointers." Generally, this involves the use of a template class that implements `operator->()` like this:

```
template<class T> class SmartPtr {
    ...
    T *operator->();
    ...
}
```

Then, if you have a class like this,

```
class Foo {
public:
    int x;
    int bar();
};
```

A smart pointer would be used in C++ as follows:

```
SmartPtr<Foo> p = CreateFoo();    // Created somehow (not shown)
...
p->x = 3;                        // Foo::x
int y = p->bar();                 // Foo::bar
```

To wrap this in Ruby, simply tell SWIG about the `SmartPtr` class and the low-level `Foo` object. Make sure you instantiate `SmartPtr` using `%template` if necessary. For example:

```
%module example
...
%template(SmartPtrFoo) SmartPtr<Foo>;
...
```

Now, in Ruby, everything should just "work":

```
irb(main):001:0> p = Example::CreateFoo()           # Create a smart-pointer somehow
#<Example::SmartPtrFoo:0x4016f4df>
irb(main):002:0> p.x = 3                           # Foo::x
3
irb(main):003:0> p.bar()                           # Foo::bar
```

If you ever need to access the underlying pointer returned by `operator->()` itself, simply use the `__deref__()` method. For example:

```
irb(main):004:0> f = p.__deref__()                 # Returns underlying Foo *
```

## 27.3.14 Cross-Language Polymorphism

SWIG's Ruby module supports cross-language polymorphism (a.k.a. the "directors" feature) similar to that for SWIG's Python module. Rather than duplicate the information presented in the [Python](#) chapter, this section just notes the differences that you need to be aware of when using this feature with Ruby.

### 27.3.14.1 Exception Unrolling

Whenever a C++ director class routes one of its virtual member function calls to a Ruby instance method, there's always the possibility that an exception will be raised in the Ruby code. By default, those exceptions are ignored, which simply means that the exception will be exposed to the Ruby interpreter. If you would like to change this behavior, you can use the `%feature("director:except")` directive to indicate what action should be taken when a Ruby exception is raised. The following code should suffice in most cases:

```
%feature("director:except") {
    throw Swig::DirectorMethodException($error);
}
```

When this feature is activated, the call to the Ruby instance method is "wrapped" using the `rb_rescue2()` function from Ruby's C API. If any Ruby exception is raised, it will be caught here and a C++ exception is raised in its place.

## 27.4 Input and output parameters

A common problem in some C programs is handling parameters passed as simple pointers. For example:

```
void add(int x, int y, int *result) {
    *result = x + y;
}
or
int sub(int *x, int *y) {
    return *x-*y;
}
```

The easiest way to handle these situations is to use the `typemaps.i` file. For example:

```
%module Example
#include "typemaps.i"

void add(int, int, int *OUTPUT);
```

```
int sub(int *INPUT, int *INPUT);
```

In Ruby, this allows you to pass simple values. For example:

```
a = Example.add(3,4)
puts a
7
b = Example.sub(7,4)
puts b
3
```

Notice how the INPUT parameters allow integer values to be passed instead of pointers and how the OUTPUT parameter creates a return result.

If you don't want to use the names INPUT or OUTPUT, use the %apply directive. For example:

```
%module Example
#include "typemaps.i"

%apply int *OUTPUT { int *result };
%apply int *INPUT { int *x, int *y};

void add(int x, int y, int *result);
int sub(int *x, int *y);
```

If a function mutates one of its parameters like this,

```
void negate(int *x) {
    *x = -(*x);
}
```

you can use INOUT like this:

```
%include "typemaps.i"
...
void negate(int *INOUT);
```

In Ruby, a mutated parameter shows up as a return value. For example:

```
a = Example.negate(3)
print a
-3
```

The most common use of these special typemap rules is to handle functions that return more than one value. For example, sometimes a function returns a result as well as a special error code:

```
/* send message, return number of bytes sent, success code, and error_code */
int send_message(char *text, int *success, int *error_code);
```

To wrap such a function, simply use the OUTPUT rule above. For example:

```
%module example
#include "typemaps.i"
...
int send_message(char *, int *OUTPUT, int *OUTPUT);
```

When used in Ruby, the function will return an array of multiple values.

```
bytes, success, error_code = send_message("Hello World")
if not success
    print "error #{error_code} : in send_message"
```

```

else
  print "Sent", bytes
end

```

Another way to access multiple return values is to use the `%apply` rule. In the following example, the parameters `rows` and `columns` are related to SWIG as OUTPUT values through the use of `%apply`

```

%module Example
#include "typemaps.i"
%apply int *OUTPUT { int *rows, int *columns };
...
void get_dimensions(Matrix *m, int *rows, int*columns);

```

In Ruby:

```

r, c = Example.get_dimensions(m)

```

## 27.5 Simple exception handling

The SWIG `%exception` directive can be used to define a user-definable exception handler that can convert C/C++ errors into Ruby exceptions. The chapter on [Customization Features](#) contains more details, but suppose you have a C++ class like the following :

```

class DoubleArray {
private:
  int n;
  double *ptr;
public:
  // Create a new array of fixed size
  DoubleArray(int size) {
    ptr = new double[size];
    n = size;
  }
  // Destroy an array
  ~DoubleArray() {
    delete ptr;
  }
  // Return the length of the array
  int length() {
    return n;
  }

  // Get an array item and perform bounds checking.
  double getitem(int i) {
    if ((i >= 0) && (i < n))
      return ptr[i];
    else
      throw RangeError();
  }
  // Set an array item and perform bounds checking.
  void setitem(int i, double val) {
    if ((i >= 0) && (i < n))
      ptr[i] = val;
    else {
      throw RangeError();
    }
  }
};

```

Since several methods in this class can throw an exception for an out-of-bounds access, you might want to catch this in the Ruby extension by writing the following in an interface file:

```

%exception {
  try {

```



```

    $action
  }
  catch (const RangeError&) {
    static VALUE cpperror = rb_define_class("C++Error", rb_eStandardError);
    rb_raise(cpperror, "Range error.");
  }
}

class DoubleArray {
  ...
};

```

The exception handling code is inserted directly into generated wrapper functions. When an exception handler is defined, errors can be caught and used to gracefully raise a Ruby exception instead of forcing the entire program to terminate with an uncaught error.

As shown, the exception handling code will be added to every wrapper function. Because this is somewhat inefficient, you might consider refining the exception handler to only apply to specific methods like this:

```

%exception getitem {
  try {
    $action
  }
  catch (const RangeError&) {
    static VALUE cpperror = rb_define_class("C++Error", rb_eStandardError);
    rb_raise(cpperror, "Range error in getitem.");
  }
}

%exception setitem {
  try {
    $action
  }
  catch (const RangeError&) {
    static VALUE cpperror = rb_define_class("C++Error", rb_eStandardError);
    rb_raise(cpperror, "Range error in setitem.");
  }
}

```

In this case, the exception handler is only attached to methods and functions named `getitem` and `setitem`.

Since SWIG's exception handling is user-definable, you are not limited to C++ exception handling. See the chapter on [Customization Features](#) for more examples.

When raising a Ruby exception from C/C++, use the `rb_raise()` function as shown above. The first argument passed to `rb_raise()` is the exception type. You can raise a custom exception type (like the `cpperror` example shown above) or one of the built-in Ruby exception types. For a list of the standard Ruby exception classes, consult a Ruby reference such as [Programming Ruby](#).

## 27.6 Typemaps

This section describes how you can modify SWIG's default wrapping behavior for various C/C++ datatypes using the `%typemap` directive. This is an advanced topic that assumes familiarity with the Ruby C API as well as the material in the ["Typemaps"](#) chapter.

Before proceeding, it should be stressed that typemaps are not a required part of using SWIG—the default wrapping behavior is enough in most cases. Typemaps are only used if you want to change some aspect of the primitive C–Ruby interface.

## 27.6.1 What is a typemap?

A typemap is nothing more than a code generation rule that is attached to a specific C datatype. For example, to convert integers from Ruby to C, you might define a typemap like this:

```
%module example

%typemap(in) int {
    $1 = (int) NUM2INT($input);
    printf("Received an integer : %d\n", $1);
}

extern int fact(int n);
```

Typemaps are always associated with some specific aspect of code generation. In this case, the "in" method refers to the conversion of input arguments to C/C++. The datatype `int` is the datatype to which the typemap will be applied. The supplied C code is used to convert values. In this code a number of special variables prefaced by a `$` are used. The `$1` variable is placeholder for a local variable of type `int`. The `$input` variable is the input Ruby object.

When this example is compiled into a Ruby module, the following sample code:

```
require 'example'

puts Example.fact(6)
```

prints the result:

```
Received an integer : 6
720
```

In this example, the typemap is applied to all occurrences of the `int` datatype. You can refine this by supplying an optional parameter name. For example:

```
%module example

%typemap(in) int n {
    $1 = (int) NUM2INT($input);
    printf("n = %d\n", $1);
}

extern int fact(int n);
```

In this case, the typemap code is only attached to arguments that exactly match `"int n"`.

The application of a typemap to specific datatypes and argument names involves more than simple text-matching—typemaps are fully integrated into the SWIG type-system. When you define a typemap for `int`, that typemap applies to `int` and qualified variations such as `const int`. In addition, the typemap system follows `typedef` declarations. For example:

```
%typemap(in) int n {
    $1 = (int) NUM2INT($input);
    printf("n = %d\n", $1);
}

typedef int Integer;
extern int fact(Integer n);    // Above typemap is applied
```

However, the matching of `typedef` only occurs in one direction. If you defined a typemap for `Integer`, it is not applied to arguments of type `int`.

Typemaps can also be defined for groups of consecutive arguments. For example:

```
%typemap(in) (char *str, int len) {
    $1 = STR2CSTR($input);
    $2 = (int) RSTRING($input)->len;
};

int count(char c, char *str, int len);
```

When a multi-argument typemap is defined, the arguments are always handled as a single Ruby object. This allows the function `count` to be used as follows (notice how the length parameter is omitted):

```
puts Example.count('o', 'Hello World')
2
```

## 27.6.2 Ruby typemaps

The previous section illustrated an "in" typemap for converting Ruby objects to C. A variety of different typemap methods are defined by the Ruby module. For example, to convert a C integer back into a Ruby object, you might define an "out" typemap like this:

```
%typemap(out) int {
    $result = INT2NUM($1);
}
```

The following list details all of the typemap methods that can be used by the Ruby module:

`%typemap(in)`

Converts Ruby objects to input function arguments

`%typemap(out)`

Converts return value of a C function to a Ruby object

`%typemap(varin)`

Assigns a C global variable from a Ruby object

`%typemap(varout)`

Returns a C global variable as a Ruby object

`%typemap(freearg)`

Cleans up a function argument (if necessary)

`%typemap(argout)`

Output argument processing

`%typemap(ret)`

Cleanup of function return values

`%typemap(memberin)`

Setting of structure/class member data

`%typemap(globalin)`

Setting of C global variables

`%typemap ( check )`

Checks function input values.

`%typemap ( default )`

Set a default value for an argument (making it optional).

`%typemap ( arginit )`

Initialize an argument to a value before any conversions occur.

Examples of these typemaps appears in the [section on typemap examples](#)

### 27.6.3 Typemap variables

Within a typemap, a number of special variables prefaced with a \$ may appear. A full list of variables can be found in the "[Typemaps](#)" chapter. This is a list of the most common variables:

`$1`

A C local variable corresponding to the actual type specified in the `%typemap` directive. For input values, this is a C local variable that is supposed to hold an argument value. For output values, this is the raw result that is supposed to be returned to Ruby.

`$input`

A VALUE holding a raw Ruby object with an argument or variable value.

`$result`

A VALUE that holds the result to be returned to Ruby.

`$1_name`

The parameter name that was matched.

`$1_type`

The actual C datatype matched by the typemap.

`$1_ltype`

An assignable version of the datatype matched by the typemap (a type that can appear on the left-hand-side of a C assignment operation). This type is stripped of qualifiers and may be an altered version of `$1_type`. All arguments and local variables in wrapper functions are declared using this type so that their values can be properly assigned.

`$symname`

The Ruby name of the wrapper function being created.

## 27.6.4 Useful Functions

When you write a typemap, you usually have to work directly with Ruby objects. The following functions may prove to be useful. (These functions plus many more can be found in [Programming Ruby](#), by David Thomas and Andrew Hunt.)

### 27.6.4.1 C Datatypes to Ruby Objects

```
INT2NUM(long or int)  - int to Fixnum or Bignum
INT2FIX(long or int)  - int to Fixnum (faster than INT2NUM)
CHR2FIX(char)         - char to Fixnum
rb_str_new2(char*)    - char* to String
rb_float_new(double)  - double to Float
```

### 27.6.4.2 Ruby Objects to C Datatypes

```
int NUM2INT(Numeric)
int FIX2INT(Numeric)
unsigned int NUM2UINT(Numeric)
unsigned int FIX2UINT(Numeric)
long NUM2LONG(Numeric)
long FIX2LONG(Numeric)
unsigned long FIX2ULONG(Numeric)
char NUM2CHR(Numeric or String)
char * STR2CSTR(String)
char * rb_str2cstr(String, int*length)
double NUM2DBL(Numeric)
```

### 27.6.4.3 Macros for VALUE

```
RSTRING(str)->len
```

length of the Ruby string

```
RSTRING(str)->ptr
```

pointer to string storage

```
RARRAY(arr)->len
```

length of the Ruby array

```
RARRAY(arr)->capa
```

capacity of the Ruby array

```
RARRAY(arr)->ptr
```

pointer to array storage

### 27.6.4.4 Exceptions

```
void rb_raise(VALUE exception, const char *fmt, ...)
```

Raises an exception. The given format string *fmt* and remaining arguments are interpreted as with `printf()`.

```
void rb_fatal(const char *fmt, ...)
```

Raises a fatal exception, terminating the process. No rescue blocks are called, but ensure blocks will be called. The given format string *fmt* and remaining arguments are interpreted as with `printf()`.

```
void rb_bug(const char *fmt, ...)
```

Terminates the process immediately — no handlers of any sort will be called. The given format string *fmt* and remaining arguments are interpreted as with `printf()`. You should call this function only if a fatal bug has been exposed.

```
void rb_sys_fail(const char *msg)
```

Raises a platform-specific exception corresponding to the last known system error, with the given string *msg*.

```
VALUE rb_rescue(VALUE (*body)(VALUE), VALUE args, VALUE(*rescue)(VALUE, VALUE), VALUE rargs)
```

Executes *body* with the given *args*. If a `StandardError` exception is raised, then execute *rescue* with the given *rargs*.

```
VALUE rb_ensure(VALUE(*body)(VALUE), VALUE args, VALUE(*ensure)(VALUE), VALUE eargs)
```

Executes *body* with the given *args*. Whether or not an exception is raised, execute *ensure* with the given *rargs* after *body* has completed.

```
VALUE rb_protect(VALUE (*body)(VALUE), VALUE args, int *result)
```

Executes *body* with the given *args* and returns nonzero in result if any exception was raised.

```
void rb_notimplement()
```

Raises a `NotImpError` exception to indicate that the enclosed function is not implemented yet, or not available on this platform.

```
void rb_exit(int status)
```

Exits Ruby with the given *status*. Raises a `SystemExit` exception and calls registered exit functions and finalizers.

```
void rb_warn(const char *fmt, ...)
```

Unconditionally issues a warning message to standard error. The given format string *fmt* and remaining arguments are interpreted as with `printf()`.

```
void rb_warning(const char *fmt, ...)
```

Conditionally issues a warning message to standard error if Ruby was invoked with the `-w` flag. The given format string *fmt* and remaining arguments are interpreted as with `printf()`.

### 27.6.4.5 Iterators

```
void rb_iter_break()
```

Breaks out of the enclosing iterator block.

```
VALUE rb_each(VALUE obj)
```

Invokes the each method of the given *obj*.

```
VALUE rb_yield(VALUE arg)
```

Transfers execution to the iterator block in the current context, passing *arg* as an argument. Multiple values may be passed in an array.

```
int rb_block_given_p()
```

Returns true if *yield* would execute a block in the current context; that is, if a code block was passed to the current method and is available to be called.

```
VALUE rb_iterate(VALUE (*method)(VALUE), VALUE args, VALUE (*block)(VALUE, VALUE),
VALUE arg2)
```

Invokes *method* with argument *args* and block *block*. A *yield* from that method will invoke *block* with the argument given to *yield*, and a second argument *arg2*.

```
VALUE rb_catch(const char *tag, VALUE (*proc)(VALUE, VALUE), VALUE value)
```

Equivalent to Ruby's *catch*.

```
void rb_throw(const char *tag, VALUE value)
```

Equivalent to Ruby's *throw*.

## 27.6.5 Typemap Examples

This section includes a few examples of typemaps. For more examples, you might look at the examples in the `Example/ruby` directory.

### 27.6.6 Converting a Ruby array to a char \*\*

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Ruby Array instance to be used as a `char **` object.

```
%module argv

// This tells SWIG to treat char ** as a special case
%typemap(in) char ** {
    /* Get the length of the array */
    int size = RARRAY($input)->len;
    int i;
    $1 = (char **) malloc((size+1)*sizeof(char *));
    /* Get the first element in memory */
    VALUE *ptr = RARRAY($input)->ptr;
    for (i=0; i < size; i++, ptr++)
        /* Convert Ruby Object String to char* */
        $1[i] = STR2CSTR(*ptr);
    $1[i]=NULL; /* End of list */
}

// This cleans up the char ** array created before
// the function call

%typemap(freearg) char ** {
    free((char *) $1);
}

// Now a test function
```

```
%inline %{
int print_args(char **argv) {
    int i = 0;
    while (argv[i]) {
        printf("argv[%d] = %s\n", i, argv[i]);
        i++;
    }
    return i;
}
%}
```

When this module is compiled, the wrapped C function now operates as follows :

```
require 'Argv'
Argv.print_args([ "Dave", "Mike", "Mary", "Jane", "John" ])
argv[0] = Dave
argv[1] = Mike
argv[2] = Mary
argv[3] = Jane
argv[4] = John
```

In the example, two different typemaps are used. The "in" typemap is used to receive an input argument and convert it to a C array. Since dynamic memory allocation is used to allocate memory for the array, the "freearg" typemap is used to later release this memory after the execution of the C function.

### 27.6.7 Collecting arguments in a hash

Ruby's solution to the "keyword arguments" capability of some other languages is to allow the programmer to pass in one or more key-value pairs as arguments to a function. All of those key-value pairs are collected in a single Hash argument that's presented to the function. If it makes sense, you might want to provide similar functionality for your Ruby interface. For example, suppose you'd like to wrap this C function that collects information about people's vital statistics:

```
void setVitalStats(const char *person, int nattributes, const char **names, int *values);
```

and you'd like to be able to call it from Ruby by passing in an arbitrary number of key-value pairs as inputs, e.g.

```
setVitalStats("Fred",
              'weight' => 270,
              'age' => 42
            )
```

To make this work, you need to write a typemap that expects a Ruby Hash as its input and somehow extracts the last three arguments (*nattributes*, *names* and *values*) needed by your C function. Let's start with the basics:

```
%typemap(in) (int nattributes, const char **names, const int *values)
              (VALUE keys_arr, int i, VALUE key, VALUE val) {
}
```

This %typemap directive tells SWIG that we want to match any function declaration that has the specified types and names of arguments somewhere in the argument list. The fact that we specified the argument names (*nattributes*, *names* and *values*) in our typemap is significant; this ensures that SWIG won't try to apply this typemap to *other* functions it sees that happen to have a similar declaration with different argument names. The arguments that appear in the second set of parentheses (*keys\_arr*, *i*, *key* and *val*) define local variables that our typemap will need.

Since we expect the input argument to be a Hash, let's next add a check for that:

```
%typemap(in) (int nattributes, const char **names, const int *values)
              (VALUE keys_arr, int i, VALUE key, VALUE val) {
    Check_Type($input, T_HASH);
```



```
}
```

`Check_Type()` is just a macro (defined in the Ruby header files) that confirms that the input argument is of the correct type; if it isn't, an exception will be raised.

The next task is to determine how many key–value pairs are present in the hash; we'll assign this number to the first typemap argument (\$1). This is a little tricky since the Ruby/C API doesn't provide a public function for querying the size of a hash, but we can get around that by calling the hash's *size* method directly and converting its result to a C `int` value:

```
%typemap(in) (int nattributes, const char **names, const int *values)
                (VALUE keys_arr, int i, VALUE key, VALUE val) {
    Check_Type($input, T_HASH);
    $1 = NUM2INT(rb_funcall($input, rb_intern("size"), 0, NULL));
}
```

So now we know the number of attributes. Next we need to initialize the second and third typemap arguments (i.e. the two C arrays) to `NULL` and set the stage for extracting the keys and values from the hash:

```
%typemap(in) (int nattributes, const char **names, const int *values)
                (VALUE keys_arr, int i, VALUE key, VALUE val) {
    Check_Type($input, T_HASH);
    $1 = NUM2INT(rb_funcall($input, rb_intern("size"), 0, NULL));
    $2 = NULL;
    $3 = NULL;
    if ($1 > 0) {
        $2 = (char **) malloc($1*sizeof(char *));
        $3 = (int *) malloc($1*sizeof(int));
    }
}
```

There are a number of ways we could extract the keys and values from the input hash, but the simplest approach is to first call the hash's *keys* method (which returns a Ruby array of the keys) and then start looping over the elements in that array:

```
%typemap(in) (int nattributes, const char **names, const int *values)
                (VALUE keys_arr, int i, VALUE key, VALUE val) {
    Check_Type($input, T_HASH);
    $1 = NUM2INT(rb_funcall($input, rb_intern("size"), 0, NULL));
    $2 = NULL;
    $3 = NULL;
    if ($1 > 0) {
        $2 = (char **) malloc($1*sizeof(char *));
        $3 = (int *) malloc($1*sizeof(int));
        keys_arr = rb_funcall($input, rb_intern("keys"), 0, NULL);
        for (i = 0; i < $1; i++) {
        }
    }
}
```

Recall that *keys\_arr* and *i* are local variables for this typemap. For each element in the *keys\_arr* array, we want to get the key itself, as well as the value corresponding to that key in the hash:

```
%typemap(in) (int nattributes, const char **names, const int *values)
                (VALUE keys_arr, int i, VALUE key, VALUE val) {
    Check_Type($input, T_HASH);
    $1 = NUM2INT(rb_funcall($input, rb_intern("size"), 0, NULL));
    $2 = NULL;
    $3 = NULL;
    if ($1 > 0) {
        $2 = (char **) malloc($1*sizeof(char *));
        $3 = (int *) malloc($1*sizeof(int));
```

```

keys_arr = rb_funcall($input, rb_intern("keys"), 0, NULL);
for (i = 0; i < $1; i++) {
    key = rb_ary_entry(keys_arr, i);
    val = rb_hash_aref($input, key);
}
}
}

```

To be safe, we should again use the `Check_Type()` macro to confirm that the key is a `String` and the value is a `Fixnum`:

```

%typemap(in) (int nattributes, const char **names, const int *values)
(VALUE keys_arr, int i, VALUE key, VALUE val) {
    Check_Type($input, T_HASH);
    $1 = NUM2INT(rb_funcall($input, rb_intern("size"), 0, NULL));
    $2 = NULL;
    $3 = NULL;
    if ($1 > 0) {
        $2 = (char **) malloc($1*sizeof(char *));
        $3 = (int *) malloc($1*sizeof(int));
        keys_arr = rb_funcall($input, rb_intern("keys"), 0, NULL);
        for (i = 0; i < $1; i++) {
            key = rb_ary_entry(keys_arr, i);
            val = rb_hash_aref($input, key);
            Check_Type(key, T_STRING);
            Check_Type(val, T_FIXNUM);
        }
    }
}

```

Finally, we can convert these Ruby objects into their C equivalents and store them in our local C arrays:

```

%typemap(in) (int nattributes, const char **names, const int *values)
(VALUE keys_arr, int i, VALUE key, VALUE val) {
    Check_Type($input, T_HASH);
    $1 = NUM2INT(rb_funcall($input, rb_intern("size"), 0, NULL));
    $2 = NULL;
    $3 = NULL;
    if ($1 > 0) {
        $2 = (char **) malloc($1*sizeof(char *));
        $3 = (int *) malloc($1*sizeof(int));
        keys_arr = rb_funcall($input, rb_intern("keys"), 0, NULL);
        for (i = 0; i < $1; i++) {
            key = rb_ary_entry(keys_arr, i);
            val = rb_hash_aref($input, key);
            Check_Type(key, T_STRING);
            Check_Type(val, T_FIXNUM);
            $2[i] = STR2CSTR(key);
            $3[i] = NUM2INT(val);
        }
    }
}

```

We're not done yet. Since we used `malloc()` to dynamically allocate the memory used for the *names* and *values* arguments, we need to provide a corresponding "freearg" typemap to free that memory so that there is no memory leak. Fortunately, this typemap is a lot easier to write:

```

%typemap(freearg) (int nattributes, const char **names, const int *values) {
    free((void *) $2);
    free((void *) $3);
}

```

All of the code for this example, as well as a sample Ruby program that uses the extension, can be found in the `Examples/ruby/hashargs` directory of the SWIG distribution.

## 27.6.8 Pointer handling

Occasionally, it might be necessary to convert pointer values that have been stored using the SWIG typed-pointer representation. Since there are several ways in which pointers can be represented, the following two functions are used to safely perform this conversion:

```
int SWIG_ConvertPtr(VALUE obj, void **ptr, swig_type_info *ty, int flags)
```

Converts a Ruby object *obj* to a C pointer whose address is *ptr* (i.e. *ptr* is a pointer to a pointer). The third argument, *ty*, is a pointer to a SWIG type descriptor structure. If *ty* is not NULL, that type information is used to validate type compatibility and other aspects of the type conversion. If *flags* is non-zero, any type errors encountered during this validation result in a Ruby `TypeError` exception being raised; if *flags* is zero, such type errors will cause `SWIG_ConvertPtr()` to return -1 but not raise an exception. If *ty* is NULL, no type-checking is performed.

```
VALUE SWIG_NewPointerObj(void *ptr, swig_type_info *ty, int own)
```

Creates a new Ruby pointer object. Here, *ptr* is the pointer to convert, *ty* is the SWIG type descriptor structure that describes the type, and *own* is a flag that indicates whether or not Ruby should take ownership of the pointer (i.e. whether Ruby should free this data when the corresponding Ruby instance is garbage-collected).

Both of these functions require the use of a special SWIG type-descriptor structure. This structure contains information about the mangled name of the datatype, type-equivalence information, as well as information about converting pointer values under C++ inheritance. For a type of `Foo *`, the type descriptor structure is usually accessed as follows:

```
Foo *foo;
SWIG_ConvertPtr($input, (void **) &foo, SWIGTYPE_p_Foo, 1);

VALUE obj;
obj = SWIG_NewPointerObj(f, SWIGTYPE_p_Foo, 0);
```

In a typemap, the type descriptor should always be accessed using the special typemap variable `$l_descriptor`. For example:

```
%typemap(in) Foo * {
    SWIG_ConvertPtr($input, (void **) &$l, $l_descriptor, 1);
}
```

### 27.6.8.1 Ruby Datatype Wrapping

```
VALUE Data_Wrap_Struct(VALUE class, void (*mark)(void *), void (*free)(void *), void *ptr)
```

Given a pointer *ptr* to some C data, and the two garbage collection routines for this data (*mark* and *free*), return a VALUE for the Ruby object.

```
VALUE Data_Make_Struct(VALUE class, c-type, void (*mark)(void *), void (*free)(void *), c-type *ptr)
```

Allocates a new instance of a C data type *c-type*, assigns it to the pointer *ptr*, then wraps that pointer with `Data_Wrap_Struct()` as above.

```
Data_Get_Struct(VALUE obj, c-type, c-type *ptr)
```

Retrieves the original C pointer of type *c-type* from the data object *obj* and assigns that pointer to *ptr*.

## 27.7 Operator overloading

SWIG allows operator overloading with, by using the `%extend` or `%rename` commands in SWIG and the following operator names (derived from Python):

```

General
__repr__      - inspect
__str__       - to_s
__cmp__       - <=>
__hash__      - hash
__nonzero__   - nonzero?

Callable
__call__      - call

Collection
__len__       - length
__getitem__   - []
__setitem__   - []=

Numeric
__add__       - +
__sub__       - -
__mul__       - *
__div__       - /
__mod__       - %
__divmod__    - divmod
__pow__       - **
__lshift__    - <<
__rshift__    - >>
__and__       - &
__xor__       - ^
__or__        - |
__neg__       - -@
__pos__       - +@
__abs__       - abs
__invert__    - ~
__int__       - to_i
__float__     - to_f
__coerce__    - coerce

Additions in 1.3.13
__lt__        - <
__le__        - <=
__eq__        - ==
__gt__        - >
__ge__        - >=

```

Note that although SWIG supports the `__eq__` magic method name for defining an equivalence operator, there is no separate method for handling *inequality* since Ruby parses the expression `a != b` as `!(a == b)`.

### 27.7.1 Example: STL Vector to Ruby Array

*FIXME: This example is out of place here!*

Another use for macros and type maps is to create a Ruby array from a STL vector of pointers. In essence, copy of all the pointers in the vector into a Ruby array. The use of the macro is to make the typemap so generic that any vector with pointers can use the type map. The following is an example of how to construct this type of macro/typemap and should give insight into constructing similar typemaps for other STL structures:

```

#define PTR_VECTOR_TO_RUBY_ARRAY(vectorclassname, classname)
%typemap(ruby, out) vectorclassname &, const vectorclassname & {

```

```

VALUE arr = rb_ary_new2($1->size());
vectorclassname::iterator i = $1->begin(), iend = $1->end();
for ( ; i!=iend; i++ )
    rb_ary_push(arr, Data_Wrap_Struct(c ## classname.klass, 0, 0, *i));
$result = arr;
}
%typemap(ruby, out) vectorclassname, const vectorclassname {
    VALUE arr = rb_ary_new2($1.size());
    vectorclassname::iterator i = $1.begin(), iend = $1.end();
    for ( ; i!=iend; i++ )
        rb_ary_push(arr, Data_Wrap_Struct(c ## classname.klass, 0, 0, *i));
    $result = arr;
}
%enddef

```

Note, that the "c ## classname.klass" is used in the preprocessor step to determine the actual object from the class name.

To use the macro with a class Foo, the following is used:

```
PTR_VECTOR_TO_RUBY_ARRAY(vector<foo *">, Foo)
```

It is also possible to create a STL vector of Ruby objects:

```

#define RUBY_ARRAY_TO_PTR_VECTOR(vectorclassname, classname)
%typemap(ruby, in) vectorclassname &, const vectorclassname & {
    Check_Type($input, T_ARRAY);
    vectorclassname *vec = new vectorclassname;
    int len = RARRAY($input)->len;
    for (int i=0; i!=len; i++) {
        VALUE inst = rb_ary_entry($input, i);
        //The following _should_ work but doesn't on HP/UX
        //    Check_Type(inst, T_DATA);
        classname *element = NULL;
        Data_Get_Struct(inst, classname, element);
        vec->push_back(element);
    }
    $1 = vec;
}

%typemap(ruby, freearg) vectorclassname &, const vectorclassname & {
    delete $1;
}
%enddef

```

It is also possible to create a Ruby array from a vector of static data types:

```

#define VECTOR_TO_RUBY_ARRAY(vectorclassname, classname)
%typemap(ruby, out) vectorclassname &, const vectorclassname & {
    VALUE arr = rb_ary_new2($1->size());
    vectorclassname::iterator i = $1->begin(), iend = $1->end();
    for ( ; i!=iend; i++ )
        rb_ary_push(arr, Data_Wrap_Struct(c ## classname.klass, 0, 0, &(*i)));
    $result = arr;
}
%typemap(ruby, out) vectorclassname, const vectorclassname {
    VALUE arr = rb_ary_new2($1.size());
    vectorclassname::iterator i = $1.begin(), iend = $1.end();
    for ( ; i!=iend; i++ )
        rb_ary_push(arr, Data_Wrap_Struct(c ## classname.klass, 0, 0, &(*i)));
    $result = arr;
}
%enddef

```

## 27.8 Advanced Topics

### 27.8.1 Creating Multi-Module Packages

The chapter on [Working with Modules](#) discusses the basics of creating multi-module extensions with SWIG, and in particular the considerations for sharing runtime type information among the different modules.

As an example, consider one module's interface file (`shape.i`) that defines our base class:

```
%module shape

%{
#include "Shape.h"
%}

class Shape {
protected:
    double xpos;
    double ypos;
protected:
    Shape(double x, double y);
public:
    double getX() const;
    double getY() const;
};
```

We also have a separate interface file (`circle.i`) that defines a derived class:

```
%module circle

%{
#include "Shape.h"
#include "Circle.h"
%}

// Import the base class definition from Shape module
%import shape.i

class Circle : public Shape {
protected:
    double radius;
public:
    Circle(double x, double y, double r);
    double getRadius() const;
};
```

We'll start by building the **Shape** extension module:

```
$ swig -c++ -ruby shape.i
```

SWIG generates a wrapper file named `shape_wrap.cxx`. To compile this into a dynamically loadable extension for Ruby, prepare an `extconf.rb` script using this template:

```
require 'mkmf'

# Since the SWIG runtime support library for Ruby
# depends on the Ruby library, make sure it's in the list
# of libraries.
$libs = append_library($libs, Config::CONFIG['RUBY_INSTALL_NAME'])

# Create the makefile
create_makefile('shape')
```

Run this script to create a Makefile and then type make to build the shared library:

```
$ ruby extconf.rb
creating Makefile

$ make
g++ -fPIC -g -O2 -I. -I/usr/local/lib/ruby/1.7/i686-linux \
    -I. -c shape_wrap.cxx
gcc -shared -L/usr/local/lib -o shape.so shape_wrap.o -L. \
    -lruby -lruby -lc
```

Note that depending on your installation, the outputs may be slightly different; these outputs are those for a Linux-based development environment. The end result should be a shared library (here, `shape.so`) containing the extension module code. Now repeat this process in a separate directory for the **Circle** module:

1. Run SWIG to generate the wrapper code (`circle_wrap.cxx`);
2. Write an `extconf.rb` script that your end-users can use to create a platform-specific Makefile for the extension;
3. Build the shared library for this extension by typing `make`.

Once you've built both of these extension modules, you can test them interactively in IRB to confirm that the Shape and Circle modules are properly loaded and initialized:

```
$ irb
irb(main):001:0> require 'shape'
true
irb(main):002:0> require 'circle'
true
irb(main):003:0> c = Circle::Circle.new(5, 5, 20)
#<Circle::Circle:0xa097208>
irb(main):004:0> c.kind_of? Shape::Shape
true
irb(main):005:0> c.getX()
5.0
```

## 27.8.2 Defining Aliases

It's a fairly common practice in the Ruby built-ins and standard library to provide aliases for method names. For example, `Array#size` is an alias for `Array#length`. If you'd like to provide an alias for one of your class' instance methods, one approach is to use SWIG's `%extend` directive to add a new method of the aliased name that calls the original function. For example:

```
class MyArray {
public:
    // Construct an empty array
    MyArray();

    // Return the size of this array
    size_t length() const;
};

%extend MyArray {
    // MyArray#size is an alias for MyArray#length
    size_t size() const {
        return self->length();
    }
}
```

A better solution is to instead use the `%alias` directive (unique to SWIG's Ruby module). The previous example could then be rewritten as:

```
// MyArray#size is an alias for MyArray#length
```

```
%alias MyArray::length "size";

class MyArray {
public:
    // Construct an empty array
    MyArray();

    // Return the size of this array
    size_t length() const;
};
```

Multiple aliases can be associated with a method by providing a comma-separated list of aliases to the `%alias` directive, e.g.

```
%alias MyArray::length "amount,quantity,size";
```

From an end-user's standpoint, there's no functional difference between these two approaches; i.e. they should get the same result from calling either `MyArray#size` or `MyArray#length`. However, when the `%alias` directive is used, SWIG doesn't need to generate all of the wrapper code that's usually associated with added methods like our `MyArray::size()` example.

Note that the `%alias` directive is implemented using SWIG's "features" mechanism and so the same name matching rules used for other kinds of features apply (see the chapter on ["Customization Features"](#) for more details).

### 27.8.3 Predicate Methods

Predicate methods in Ruby are those which return either `true` or `false`. By convention, these methods' names end in a question mark; some examples from built-in Ruby classes include `Array#empty?` (which returns `true` for an array containing no elements) and `Object#instance_of?` (which returns `true` if the object is an instance of the specified class). For consistency with Ruby conventions you would also want your interface's predicate methods' names to end in a question mark and return `true` or `false`.

One cumbersome solution to this problem is to rename the method (using SWIG's `%rename` directive) and provide a custom typemap that converts the function's actual return type to Ruby's `true` or `false`. For example:

```
%rename("is_it_safe?") is_it_safe();

%typemap(out) int is_it_safe
    "$result = ($1 != 0) ? Qtrue : Qfalse;";

int is_it_safe();
```

A better solution is to instead use the `%predicate` directive (unique to SWIG's Ruby module) to designate certain methods as predicate methods. For the previous example, this would look like:

```
%predicate is_it_safe();

int is_it_safe();
```

and to use this method from your Ruby code:

```
irb(main):001:0> Example::is_it_safe?
true
```

Note that the `%predicate` directive is implemented using SWIG's "features" mechanism and so the same name matching rules used for other kinds of features apply (see the chapter on ["Customization Features"](#) for more details).

### 27.8.4 Specifying Mixin Modules

The Ruby language doesn't support multiple inheritance, but it does allow you to mix one or more modules into a class using Ruby's `include` method. For example, if you have a Ruby class that defines an *each* instance method, e.g.



```
class Set
  def initialize
    @members = []
  end

  def each
    @members.each { |m| yield m }
  end
end
```

then you can mix-in Ruby's `Enumerable` module to easily add a lot of functionality to your class:

```
class Set
  include Enumerable

  def initialize
    @members = []
  end

  def each
    @members.each { |m| yield m }
  end
end
```

To get the same benefit for your SWIG-wrapped classes, you can use the `%mixin` directive to specify the names of one or more modules that should be mixed-in to a class. For the above example, the SWIG interface specification might look like this:

```
%mixin Set "Enumerable";

class Set {
public:
  // Constructor
  Set();

  // Iterates through set members
  void each();
};
```

Multiple modules can be mixed into a class by providing a comma-separated list of module names to the `%mixin` directive, e.g.

```
%mixin Set "Fee,Fi,Fo,Fum";
```

Note that the `%mixin` directive is implemented using SWIG's "features" mechanism and so the same name matching rules used for other kinds of features apply (see the chapter on ["Customization Features"](#) for more details).

## 27.8.5 Interacting with Ruby's Garbage Collector

**This section is still unfinished!**

By default, SWIG ensures that any C++ objects it creates are destroyed when the corresponding Ruby instance is garbage-collected. For example, if you have an interface like this:

```
class Foo
{
public:
  // Construct a new Foo object
  Foo();
};
```

When a user of this extension creates a new `Foo` instance from Ruby, it will construct a new C++ `Foo` object behind the scenes, and when that Ruby instance is garbage-collected, the same C++ object will be destroyed.

But in the real world, things aren't always that simple.

It is often the case, especially for C++ libraries, that objects contain references to other objects. For example, consider a class library that models a zoo and the animals in the zoo:

```
%module zoo

%{
#include <string>
#include <vector>

#include "zoo.h"
%}

class Animal
{
protected:
    std::string name;

public:
    // Construct an animal with this name
    Animal(const char* nm) : name(nm) {}

    // Return the animal's name
    const char* getName() const { return name.c_str(); }
};

class Zoo
{
protected:
    std::vector<animal *"> animals;

public:
    // Construct an empty zoo
    Zoo() {}

    // Add a new animal to the zoo
    void addAnimal(Animal* animal) {
        animals.push_back(animal);
    }

    // Return the number of animals in the zoo
    size_t getNumAnimals() const {
        return animals.size();
    }

    // Return a pointer to the ith animal
    Animal* getAnimal(size_t i) const {
        return animals[i];
    }
};
```

Basically, a Zoo is modeled as a "container" for animals. And we can SWIG this set of classes, and running `irb` gives the following:

```
$ irb
irb(main):001:0> require 'zoo'
true
irb(main):002:0> zoo = Zoo::Zoo.new
#<Zoo::Zoo:0xa090458>
irb(main):003:0> zoo.addAnimal(Zoo::Animal.new("Lassie"))
nil
irb(main):004:0> zoo.addAnimal(Zoo::Animal.new("Felix"))
nil
irb(main):005:0> zoo.getNumAnimals()
```

```

2
irb(main):006:0> zoo.getAnimal(0).getName()
"Lassie"
irb(main):007:0> GC.start
nil
irb(main):008:0> zoo.getAnimal(0).getName()
(irb):8: [BUG] Segmentation fault
ruby 1.7.2 (2002-03-25) [i386-cygwin]
Aborted (core dumped)

```

Observe that after the garbage collector runs (as a result of our call to `GC.start`) the call to `Animal#getName` causes a segmentation fault. To understand what went wrong requires a basic understanding of Ruby's "mark and sweep" garbage collection scheme.

*Add brief discussion of mark and sweep here?*

So the problem with our previous example is that during the GC "mark" phase, Ruby has no way of knowing that our two `Animal` instances ("Lassie" and "Felix") are still in use. As far as Ruby can tell, both of these objects are unreachable and should be garbage-collected. We'd like to fix things so that when the `Zoo` instance is visited during the "mark" phase, that it in turn marks the two animals as in use.

The Ruby/C API provides for this need by allowing extension developers to specify customized "mark" functions for data objects like our `Zoo` and `Animal` classes. This mark function takes a single argument, which is a pointer to the C++ object being marked; it should, in turn, call `rb_gc_mark()` for any instances that are reachable from the current object. The mark function for our `Zoo` class should therefore loop over all of the animals in the zoo and call `rb_gc_mark()` for each of the Ruby instances associated with those C++ `Animal` objects:

```

void Zoo_markfunc(void *ptr)
{
    Animal *cppAnimal;
    VALUE  rubyAnimal;
    Zoo *zoo;

    zoo = static_cast<zoo>
    *=">(ptr);
    for (size_t i = 0; i < zoo->getNumAnimals(); i++) {
        cppAnimal = zoo->getAnimal(i);
        rubyAnimal = SWIG_RubyInstanceFor(cppAnimal);
        rb_gc_mark(rubyAnimal);
    }
}

```

*SWIG\_RubyInstanceFor()* is an imaginary function that takes a pointer to a C/C++ object as its input and returns a `VALUE` corresponding to the Ruby instance that wraps this object. Currently, SWIG doesn't keep track of this kind of mapping at all.

You can use the `%markfunc` directive to associate the name of this function with a SWIGed class:

```
%markfunc Zoo "Zoo_markfunc";
```

Note that the `%markfunc` and `%freefunc` directives are implemented using SWIG's "features" mechanism and so the same name matching rules used for other kinds of features apply (see the chapter on ["Customization Features"](#)) for more details).

## 28 SWIG and Tcl

- [Preliminaries](#)
  - ◆ [Getting the right header files](#)
  - ◆ [Compiling a dynamic module](#)
  - ◆ [Static linking](#)
  - ◆ [Using your module](#)
  - ◆ [Compilation of C++ extensions](#)
  - ◆ [Compiling for 64-bit platforms](#)
  - ◆ [Setting a package prefix](#)
  - ◆ [Using namespaces](#)
- [Building Tcl/Tk Extensions under Windows 95/NT](#)
  - ◆ [Running SWIG from Developer Studio](#)
  - ◆ [Using NMAKE](#)
- [A tour of basic C/C++ wrapping](#)
  - ◆ [Modules](#)
  - ◆ [Functions](#)
  - ◆ [Global variables](#)
  - ◆ [Constants and enums](#)
  - ◆ [Pointers](#)
  - ◆ [Structures](#)
  - ◆ [C++ classes](#)
  - ◆ [C++ inheritance](#)
  - ◆ [Pointers, references, values, and arrays](#)
  - ◆ [C++ overloaded functions](#)
  - ◆ [C++ operators](#)
  - ◆ [C++ namespaces](#)
  - ◆ [C++ templates](#)
  - ◆ [C++ Smart Pointers](#)
- [Further details on the Tcl class interface](#)
  - ◆ [Proxy classes](#)
  - ◆ [Memory management](#)
- [Input and output parameters](#)
- [Exception handling](#)
- [Typemaps](#)
  - ◆ [What is a typemap?](#)
  - ◆ [Tcl typemaps](#)
  - ◆ [Typemap variables](#)
  - ◆ [Converting a Tcl list to a char \\*\\*](#)
  - ◆ [Returning values in arguments](#)
  - ◆ [Useful functions](#)
  - ◆ [Standard typemaps](#)
  - ◆ [Pointer handling](#)
- [Turning a SWIG module into a Tcl Package.](#)
- [Building new kinds of Tcl interfaces \(in Tcl\)](#)
  - ◆ [Proxy classes](#)

**Caution: This chapter is under repair!**

This chapter discusses SWIG's support of Tcl. SWIG currently requires Tcl 8.0 or a later release. Earlier releases of SWIG supported Tcl 7.x, but this is no longer supported.

## 28.1 Preliminaries

To build a Tcl module, run SWIG using the `-tcl` option :

```
$ swig -tcl example.i
```

If building a C++ extension, add the `-c++` option:

```
$ swig -c++ -tcl example.i
```

This creates a file `example_wrap.c` or `example_wrap.cxx` that contains all of the code needed to build a Tcl extension module. To finish building the module, you need to compile this file and link it with the rest of your program.

### 28.1.1 Getting the right header files

In order to compile the wrapper code, the compiler needs the `tcl.h` header file. This file is usually contained in the directory

```
/usr/local/include
```

Be aware that some Tcl versions install this header file with a version number attached to it. If this is the case, you should probably make a symbolic link so that `tcl.h` points to the correct header file.

### 28.1.2 Compiling a dynamic module

The preferred approach to building an extension module is to compile it into a shared object file or DLL. To do this, you will need to compile your program using commands like this (shown for Linux):

```
$ swig -tcl example.i
$ gcc -c example.c
$ gcc -c example_wrap.c -I/usr/local/include
$ gcc -shared example.o example_wrap.o -o example.so
```

The exact commands for doing this vary from platform to platform. SWIG tries to guess the right options when it is installed. Therefore, you may want to start with one of the examples in the `SWIG/Examples/tcl` directory. If that doesn't work, you will need to read the man-pages for your compiler and linker to get the right set of options. You might also check the [SWIG Wiki](#) for additional information.

When linking the module, the name of the output file has to match the name of the module. If the name of your SWIG module is "example", the name of the corresponding object file should be "example.so". The name of the module is specified using the `%module` directive or the `-module` command line option.

### 28.1.3 Static linking

An alternative approach to dynamic linking is to rebuild the Tcl interpreter with your extension module added to it. In the past, this approach was sometimes necessary due to limitations in dynamic loading support on certain machines. However, the situation has improved greatly over the last few years and you should not consider this approach unless there is really no other option.

The usual procedure for adding a new module to Tcl involves writing a special function `Tcl_AppInit()` and using it to initialize the interpreter and your module. With SWIG, the `tclsh.i` and `wish.i` library files can be used to rebuild the `tclsh` and `wish` interpreters respectively. For example:

```
%module example

extern int fact(int);
extern int mod(int, int);
extern double My_variable;
```

```
%include tclsh.i          // Include code for rebuilding tclsh
```

The `tclsh.i` library file includes supporting code that contains everything needed to rebuild `tclsh`. To rebuild the interpreter, you simply do something like this:

```
$ swig -tcl example.i
$ gcc example.c example_wrap.c \
    -Xlinker -export-dynamic \
    -DHAVE_CONFIG_H -I/usr/local/include/ \
    -L/usr/local/lib -ltcl -lm -ldl \
    -o mytclsh
```

You will need to supply the same libraries that were used to build Tcl the first time. This may include system libraries such as `-lsocket`, `-lnsl`, and `-lpthread`. If this actually works, the new version of Tcl should be identical to the default version except that your extension module will be a built-in part of the interpreter.

**Comment:** In practice, you should probably try to avoid static linking if possible. Some programmers may be inclined to use static linking in the interest of getting better performance. However, the performance gained by static linking tends to be rather minimal in most situations (and quite frankly not worth the extra hassle in the opinion of this author).

## 28.1.4 Using your module

To use your module, simply use the Tcl `load` command. If all goes well, you will be able to this:

```
$ tclsh
% load ./example.so
% fact 4
24
%
```

A common error received by first-time users is the following:

```
% load ./example.so
couldn't find procedure Example_Init
%
```

This error is almost always caused when the name of the shared object file doesn't match the name of the module supplied using the SWIG `%module` directive. Double-check the interface to make sure the module name and the shared object file match. Another possible cause of this error is forgetting to link the SWIG-generated wrapper code with the rest of your application when creating the extension module.

Another common error is something similar to the following:

```
% load ./example.so
couldn't load file "./example.so": ./example.so: undefined symbol: fact
%
```

This error usually indicates that you forgot to include some object files or libraries in the linking of the shared library file. Make sure you compile both the SWIG wrapper file and your original program into a shared library file. Make sure you pass all of the required libraries to the linker.

Sometimes unresolved symbols occur because a wrapper has been created for a function that doesn't actually exist in a library. This usually occurs when a header file includes a declaration for a function that was never actually implemented or it was removed from a library without updating the header file. To fix this, you can either edit the SWIG input file to remove the offending declaration or you can use the `%ignore` directive to ignore the declaration.

Finally, suppose that your extension module is linked with another library like this:

```
$ gcc -shared example.o example_wrap.o -L/home/beazley/projects/lib -lfoo \
-o example.so
```

If the `foo` library is compiled as a shared library, you might get the following problem when you try to use your module:

```
% load ./example.so
couldn't load file "./example.so": libfoo.so: cannot open shared object file:
No such file or directory
%
```

This error is generated because the dynamic linker can't locate the `libfoo.so` library. When shared libraries are loaded, the system normally only checks a few standard locations such as `/usr/lib` and `/usr/local/lib`. To fix this problem, there are several things you can do. First, you can recompile your extension module with extra path information. For example, on Linux you can do this:

```
$ gcc -shared example.o example_wrap.o -L/home/beazley/projects/lib -lfoo \
-Xlinker -rpath /home/beazley/projects/lib \
-o example.so
```

Alternatively, you can set the `LD_LIBRARY_PATH` environment variable to include the directory with your shared libraries. If setting `LD_LIBRARY_PATH`, be aware that setting this variable can introduce a noticeable performance impact on all other applications that you run. To set it only for Tcl, you might want to do this instead:

```
$ env LD_LIBRARY_PATH=/home/beazley/projects/lib tclsh
```

Finally, you can use a command such as `ldconfig` to add additional search paths to the default system configuration (this requires root access and you will need to read the man pages).

## 28.1.5 Compilation of C++ extensions

Compilation of C++ extensions has traditionally been a tricky problem. Since the Tcl interpreter is written in C, you need to take steps to make sure C++ is properly initialized and that modules are compiled correctly.

On most machines, C++ extension modules should be linked using the C++ compiler. For example:

```
% swig -c++ -tcl example.i
% g++ -c example.cxx
% g++ -c example_wrap.cxx -I/usr/local/include
% g++ -shared example.o example_wrap.o -o example.so
```

In addition to this, you may need to include additional library files to make it work. For example, if you are using the Sun C++ compiler on Solaris, you often need to add an extra library `-lCrun` like this:

```
% swig -c++ -tcl example.i
% CC -c example.cxx
% CC -c example_wrap.cxx -I/usr/local/include
% CC -G example.o example_wrap.o -L/opt/SUNWspro/lib -o example.so -lCrun
```

Of course, the extra libraries to use are completely non-portable—you will probably need to do some experimentation.

Sometimes people have suggested that it is necessary to relink the Tcl interpreter using the C++ compiler to make C++ extension modules work. In the experience of this author, this has never actually appeared to be necessary. Relinking the interpreter with C++ really only includes the special run-time libraries described above—as long as you link your extension modules with these libraries, it should not be necessary to rebuild Tcl.

If you aren't entirely sure about the linking of a C++ extension, you might look at an existing C++ program. On many Unix machines, the `ldd` command will list library dependencies. This should give you some clues about what you might have to

include when you link your extension module. For example:

```
$ ldd swig
    libstdc++-libc6.1-1.so.2 => /usr/lib/libstdc++-libc6.1-1.so.2 (0x40019000)
    libm.so.6 => /lib/libm.so.6 (0x4005b000)
    libc.so.6 => /lib/libc.so.6 (0x40077000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

As a final complication, a major weakness of C++ is that it does not define any sort of standard for binary linking of libraries. This means that C++ code compiled by different compilers will not link together properly as libraries nor is the memory layout of classes and data structures implemented in any kind of portable manner. In a monolithic C++ program, this problem may be unnoticed. However, in Tcl, it is possible for different extension modules to be compiled with different C++ compilers. As long as these modules are self-contained, this probably won't matter. However, if these modules start sharing data, you will need to take steps to avoid segmentation faults and other erratic program behavior. If working with lots of software components, you might want to investigate using a more formal standard such as COM.

### 28.1.6 Compiling for 64-bit platforms

On platforms that support 64-bit applications (Solaris, Irix, etc.), special care is required when building extension modules. On these machines, 64-bit applications are compiled and linked using a different set of compiler/linker options. In addition, it is not generally possible to mix 32-bit and 64-bit code together in the same application.

To utilize 64-bits, the Tcl executable will need to be recompiled as a 64-bit application. In addition, all libraries, wrapper code, and every other part of your application will need to be compiled for 64-bits. If you plan to use other third-party extension modules, they will also have to be recompiled as 64-bit extensions.

If you are wrapping commercial software for which you have no source code, you will be forced to use the same linking standard as used by that software. This may prevent the use of 64-bit extensions. It may also introduce problems on platforms that support more than one linking standard (e.g., `-o32` and `-n32` on Irix).

### 28.1.7 Setting a package prefix

To avoid namespace problems, you can instruct SWIG to append a package prefix to all of your functions and variables. This is done using the `-prefix` option as follows :

```
swig -tcl -prefix Foo example.i
```

If you have a function "bar" in the SWIG file, the prefix option will append the prefix to the name when creating a command and call it "Foo\_bar".

### 28.1.8 Using namespaces

Alternatively, you can have SWIG install your module into a Tcl namespace by specifying the `-namespace` option :

```
swig -tcl -namespace example.i
```

By default, the name of the namespace will be the same as the module name, but you can override it using the `-prefix` option.

When the `-namespace` option is used, objects in the module are always accessed with the namespace name such as `Foo::bar`.

## 28.2 Building Tcl/Tk Extensions under Windows 95/NT

Building a SWIG extension to Tcl/Tk under Windows 95/NT is roughly similar to the process used with Unix. Normally, you will want to produce a DLL that can be loaded into `tcsh` or `wish`. This section covers the process of using SWIG with Microsoft Visual C++, although the procedure may be similar with other compilers.



## 28.2.1 Running SWIG from Developer Studio

If you are developing your application within Microsoft developer studio, SWIG can be invoked as a custom build option. The process roughly follows these steps :

- Open up a new workspace and use the AppWizard to select a DLL project.
- Add both the SWIG interface file (the .i file), any supporting C files, and the name of the wrapper file that will be created by SWIG (ie. `example_wrap.c`). Note : If using C++, choose a different suffix for the wrapper file such as `example_wrap.cxx`. Don't worry if the wrapper file doesn't exist yet—Developer studio will keep a reference to it around.
- Select the SWIG interface file and go to the settings menu. Under settings, select the "Custom Build" option.
- Enter "SWIG" in the description field.
- Enter `swig -tcl -o $(ProjDir)\$(InputName)_wrap.c $(InputPath)` in the "Build command(s) field"
- Enter `$(ProjDir)\$(InputName)_wrap.c` in the "Output files(s) field".
- Next, select the settings for the entire project and go to "C++:Preprocessor". Add the include directories for your Tcl installation under "Additional include directories".
- Finally, select the settings for the entire project and go to "Link Options". Add the Tcl library file to your link libraries. For example `"tcl80.lib"`. Also, set the name of the output file to match the name of your Tcl module (ie. `example.dll`).
- Build your project.

Now, assuming all went well, SWIG will be automatically invoked when you build your project. Any changes made to the interface file will result in SWIG being automatically invoked to produce a new version of the wrapper file. To run your new Tcl extension, simply run `tclsh` or `wish` and use the `load` command. For example :

```
MSDOS > tclsh80
% load example.dll
% fact 4
24
%
```

## 28.2.2 Using NMAKE

Alternatively, SWIG extensions can be built by writing a Makefile for NMAKE. To do this, make sure the environment variables for MSVC++ are available and the MSVC++ tools are in your path. Now, just write a short Makefile like this :

```
# Makefile for building various SWIG generated extensions

SRCS      = example.c
IFILE     = example
INTERFACE = $(IFILE).i
WRAPFILE  = $(IFILE)_wrap.c

# Location of the Visual C++ tools (32 bit assumed)

TOOLS     = c:\msdev
TARGET    = example.dll
CC        = $(TOOLS)\bin\cl.exe
LINK      = $(TOOLS)\bin\link.exe
INCLUDE32 = -I$(TOOLS)\include
MACHINE   = IX86

# C Library needed to build a DLL

DLLIBS    = msvcrt.lib oldnames.lib

# Windows libraries that are apparently needed
WINLIBS   = kernel32.lib advapi32.lib user32.lib gdi32.lib comdlg32.lib
winspool.lib
```

```
# Libraries common to all DLLs
LIBS          = $(DLLIBC) $(WINLIB)

# Linker options
LOPT          = -debug:full -debugtype:cv /NODEFAULTLIB /RELEASE /NOLOGO /
MACHINE:$(MACHINE) -entry:_DllMainCRTStartup@12 -dll

# C compiler flags

CFLAGS        = /Z7 /Od /c /nologo
TCL_INCLUDES  = -Id:\tcl8.0a2\generic -Id:\tcl8.0a2\win
TCLLIB        = d:\tcl8.0a2\win\tcl80.lib

tcl::
    ../../swig -tcl -o $(WRAPFILE) $(INTERFACE)
    $(CC) $(CFLAGS) $(TCL_INCLUDES) $(SRCS) $(WRAPFILE)
    set LIB=$(TOOLS)\lib
    $(LINK) $(LOPT) -out:example.dll $(LIBS) $(TCLLIB) example.obj example_wrap.obj
```

To build the extension, run NMAKE (you may need to run vcvars32 first). This is a pretty minimal Makefile, but hopefully its enough to get you started. With a little practice, you'll be making lots of Tcl extensions.

## 28.3 A tour of basic C/C++ wrapping

By default, SWIG tries to build a very natural Tcl interface to your C/C++ code. Functions are wrapped as functions, classes are wrapped in an interface that mimics the style of Tk widgets and [incr Tcl] classes. This section briefly covers the essential aspects of this wrapping.

### 28.3.1 Modules

The SWIG `%module` directive specifies the name of the Tcl module. If you specify `%module example`, then everything is compiled into an extension module `example.so`. When choosing a module name, make sure you don't use the same name as a built-in Tcl command.

One pitfall to watch out for is module names involving numbers. If you specify a module name like `%module md5`, you'll find that the load command no longer seems to work:

```
% load ./md5.so
couldn't find procedure Md_Init
```

To fix this, supply an extra argument to load like this:

```
% load ./md5.so md5
```

### 28.3.2 Functions

Global functions are wrapped as new Tcl built-in commands. For example,

```
%module example
int fact(int n);
```

creates a built-in function `fact` that works exactly like you think it does:

```
% load ./example.so
% fact 4
24
% set x [fact 6]
%
```

### 28.3.3 Global variables

C/C++ global variables are wrapped by Tcl global variables. For example:

```
// SWIG interface file with global variables
%module example
...
extern double density;
...
```

Now look at the Tcl interface:

```
% puts $density          # Output value of C global variable
1.0
% set density 0.95        # Change value
```

If you make an error in variable assignment, you will get an error message. For example:

```
% set density "hello"
can't set "density": Type error. expected a double.
%
```

If a variable is declared as `const`, it is wrapped as a read-only variable. Attempts to modify its value will result in an error.

To make ordinary variables read-only, you can use the `%immutable` directive. For example:

```
%immutable;
extern char *path;
%mutable;
```

The `%immutable` directive stays in effect until it is explicitly disabled using `%mutable`.

If you just want to make a specific variable immutable, supply a declaration name. For example:

```
%immutable path;
...
extern char *path;      // Read-only (due to %immutable)
```

### 28.3.4 Constants and enums

C/C++ constants are installed as global Tcl variables containing the appropriate value. To create a constant, use `#define`, `enum`, or the `%constant` directive. For example:

```
#define PI 3.14159
#define VERSION "1.0"

enum Beverage { ALE, LAGER, STOUT, PILSNER };

%constant int FOO = 42;
%constant const char *path = "/usr/local";
```

For enums, make sure that the definition of the enumeration actually appears in a header file or in the wrapper file somehow---if you just stick an enum in a SWIG interface without also telling the C compiler about it, the wrapper code won't compile.

Note: declarations declared as `const` are wrapped as read-only variables and will be accessed using the `cvar` object described in the previous section. They are not wrapped as constants. For further discussion about this, see the [SWIG Basics](#) chapter.

Constants are not guaranteed to remain constant in Tcl---the value of the constant could be accidentally reassigned. You will just have to be careful.

A peculiarity of installing constants as variables is that it is necessary to use the Tcl `global` statement to access constants in procedure bodies. For example:

```
proc blah {} {
    global FOO
    bar $FOO
}
```

If a program relies on a lot of constants, this can be extremely annoying. To fix the problem, consider using the following `typemap` rule:

```
%apply int CONSTANT { int x };
#define FOO 42
...
void bar(int x);
```

When applied to an input argument, the `CONSTANT` rule allows a constant to be passed to a function using its actual value or a symbolic identifier name. For example:

```
proc blah {} {
    bar FOO
}
```

When an identifier name is given, it is used to perform an implicit hash-table lookup of the value during argument conversion. This allows the `global` statement to be omitted.

### 28.3.5 Pointers

C/C++ pointers are fully supported by SWIG. Furthermore, SWIG has no problem working with incomplete type information. Here is a rather simple interface:

```
%module example

FILE *fopen(const char *filename, const char *mode);
int fputs(const char *, FILE *);
int fclose(FILE *);
```

When wrapped, you will be able to use the functions in a natural way from Tcl. For example:

```
% load ./example.so
% set f [fopen junk w]
% fputs "Hello World\n" $f
% fclose $f
```

If this makes you uneasy, rest assured that there is no deep magic involved. Underneath the covers, pointers to C/C++ objects are simply represented as opaque values—normally an encoded character string like this:

```
% puts $f
_c0671108_p_FILE
%
```

This pointer value can be freely passed around to different C functions that expect to receive an object of type `FILE *`. The only thing you can't do is dereference the pointer from Tcl.

The `NULL` pointer is represented by the string `NULL`.

As much as you might be inclined to modify a pointer value directly from Tcl, don't. The hexadecimal encoding is not necessarily the same as the logical memory address of the underlying object. Instead it is the raw byte encoding of the pointer value. The encoding will vary depending on the native byte-ordering of the platform (i.e., big-endian vs. little-endian). Similarly, don't try to manually cast a pointer to a new type by simply replacing the type-string. This may not work like you expect and it is

particularly dangerous when casting C++ objects. If you need to cast a pointer or change its value, consider writing some helper functions instead. For example:

```
%inline %{
/* C-style cast */
Bar *FooToBar(Foo *f) {
    return (Bar *) f;
}

/* C++-style cast */
Foo *BarToFoo(Bar *b) {
    return dynamic_cast<Foo*>(b);
}

Foo *IncrFoo(Foo *f, int i) {
    return f+i;
}
%}
```

Also, if working with C++, you should always try to use the new C++ style casts. For example, in the above code, the C-style cast may return a bogus result whereas as the C++-style cast will return None if the conversion can't be performed.

### 28.3.6 Structures

If you wrap a C structure, it is wrapped by a Tcl interface that somewhat resembles a Tk widget. This provides a very natural interface. For example,

```
struct Vector {
    double x,y,z;
};
```

is used as follows:

```
% Vector v
% v configure -x 3.5 -y 7.2
% puts "[v cget -x] [v cget -y] [v cget -z]"
3.5 7.2 0.0
%
```

Similar access is provided for unions and the data members of C++ classes.

In the above example, `v` is a name that's used for the object. However, underneath the covers, there's a pointer to a raw C structure. This can be obtained by looking at the `-this` attribute. For example:

```
% puts [v cget -this]
_88e31408_p_Vector
```

Further details about the relationship between the Tcl and the underlying C structure are covered a little later.

`const` members of a structure are read-only. Data members can also be forced to be read-only using the `%immutable` directive. For example:

```
struct Foo {
    ...
    %immutable;
    int x;          /* Read-only members */
    char *name;
    %mutable;
    ...
};
```

## SWIG-1.3 Documentation

When `char *` members of a structure are wrapped, the contents are assumed to be dynamically allocated using `malloc` or `new` (depending on whether or not SWIG is run with the `-c++` option). When the structure member is set, the old contents will be released and a new value created. If this is not the behavior you want, you will have to use a `typemap` (described later).

If a structure contains arrays, access to those arrays is managed through pointers. For example, consider this:

```
struct Bar {
    int  x[16];
};
```

If accessed in Tcl, you will see behavior like this:

```
% Bar b
% puts [b cget -x]
_801861a4_p_int
%
```

This pointer can be passed around to functions that expect to receive an `int *` (just like C). You can also set the value of an array member using another pointer. For example:

```
% Bar c
% c configure -x [b cget -x]  # Copy contents of b.x to c.x
```

For array assignment, SWIG copies the entire contents of the array starting with the data pointed to by `b.x`. In this example, 16 integers would be copied. Like C, SWIG makes no assumptions about bounds checking—if you pass a bad pointer, you may get a segmentation fault or access violation.

When a member of a structure is itself a structure, it is handled as a pointer. For example, suppose you have two structures like this:

```
struct Foo {
    int a;
};

struct Bar {
    Foo f;
};
```

Now, suppose that you access the `f` attribute of `Bar` like this:

```
% Bar b
% set x [b cget -f]
```

In this case, `x` is a pointer that points to the `Foo` that is inside `b`. This is the same value as generated by this C code:

```
Bar b;
Foo *x = &b->f;      /* Points inside b */
```

However, one peculiarity of accessing a substructure like this is that the returned value does work quite like you might expect. For example:

```
% Bar b
% set x [b cget -f]
% x cget -a
invalid command name "x"
```

This is because the returned value was not created in a normal way from the interpreter (`x` is not a command object). To make it function normally, just evaluate the variable like this:

```
% Bar b
% set x [b cget -f]
```

```
% $x cget -a
0
%
```

In this example, `x` points inside the original structure. This means that modifications work just like you would expect. For example:

```
% Bar b
% set x [b cget -f]
% $x configure -a 3      # Modifies contents of f (inside b)
% [b cget -f] -configure -a 3 # Same thing
```

In many of these structure examples, a simple name like "v" or "b" has been given to wrapped structures. If necessary, this name can be passed to functions that expect to receive an object. For example, if you have a function like this,

```
void blah(Foo *f);
```

you can call the function in Tcl as follows:

```
% Foo x          # Create a Foo object
% blah x         # Pass the object to a function
```

It is also possible to call the function using the raw pointer value. For instance:

```
% blah [x cget -this] # Pass object to a function
```

It is also possible to create and use objects using variables. For example:

```
% set b [Bar]      # Create a Bar
% $b cget -f        # Member access
% puts $b
_108fea88_p_Bar
%
```

Finally, to destroy objects created from Tcl, you can either let the object name go out of scope or you can explicitly delete the object. For example:

```
% Foo f          # Create object f
% rename f ""
```

or

```
% Foo f          # Create object f
% f -delete
```

Note: Tcl only destroys the underlying object if it has ownership. See the memory management section that appears shortly.

### 28.3.7 C++ classes

C++ classes are wrapped as an extension of structure wrapping. For example, if you have this class,

```
class List {
public:
    List();
    ~List();
    int  search(char *item);
    void insert(char *item);
    void remove(char *item);
    char *get(int n);
    int  length;
```

```
};
```

you can use it in Tcl like this:

```
% List x
% x insert Ale
% x insert Stout
% x insert Lager
% x get 1
Stout
% puts [l cget -length]
3
%
```

Class data members are accessed in the same manner as C structures.

Static class members are accessed as global functions or variables. To illustrate, suppose you have a class like this:

```
class Spam {
public:
    static void foo();
    static int bar;
};
```

In Tcl, the static member is accessed as follows:

```
% Spam_foo      # Spam::foo()
% puts $Spam_bar # Spam::bar
```

### 28.3.8 C++ inheritance

SWIG is fully aware of issues related to C++ inheritance. Therefore, if you have classes like this

```
class Foo {
...
};

class Bar : public Foo {
...
};
```

An object of type Bar can be used where a Foo is expected. For example, if you have this function:

```
void spam(Foo *f);
```

then the function `spam( )` accepts a `Foo *` or a pointer to any class derived from `Foo`. For instance:

```
% Foo f      # Create a Foo
% Bar b      # Create a Bar
% spam f     # OK
% spam b     # OK
```

It is safe to use multiple inheritance with SWIG.

### 28.3.9 Pointers, references, values, and arrays

In C++, there are many different ways a function might receive and manipulate objects. For example:

```
void spam1(Foo *x);    // Pass by pointer
void spam2(Foo &x);    // Pass by reference
```



```
void spam3(Foo x);           // Pass by value
void spam4(Foo x[]);        // Array of objects
```

In Tcl, there is no detailed distinction like this. Because of this, SWIG unifies all of these types together in the wrapper code. For instance, if you actually had the above functions, it is perfectly legal to do this:

```
% Foo f           # Create a Foo
% spam1 f         # Ok. Pointer
% spam2 f         # Ok. Reference
% spam3 f         # Ok. Value.
% spam4 f         # Ok. Array (1 element)
```

Similar behavior occurs for return values. For example, if you had functions like this,

```
Foo *spam5();
Foo &spam6();
Foo spam7();
```

then all three functions will return a pointer to some `Foo` object. Since the third function (`spam7`) returns a value, newly allocated memory is used to hold the result and a pointer is returned (Tcl will release this memory when the return value is garbage collected).

### 28.3.10 C++ overloaded functions

C++ overloaded functions, methods, and constructors are mostly supported by SWIG. For example, if you have two functions like this:

```
void foo(int);
void foo(char *c);
```

You can use them in Tcl in a straightforward manner:

```
% foo 3           # foo(int)
% foo Hello       # foo(char *c)
```

Similarly, if you have a class like this,

```
class Foo {
public:
    Foo();
    Foo(const Foo &);
    ...
};
```

you can write Tcl code like this:

```
% Foo f           # Create a Foo
% Foo g f         # Copy f
```

Overloading support is not quite as flexible as in C++. Sometimes there are methods that SWIG can't disambiguate. For example:

```
void spam(int);
void spam(short);
```

or

```
void foo(Bar *b);
void foo(Bar &b);
```

If declarations such as these appear, you will get a warning message like this:

```
example.i:12: Warning(509): Overloaded spam(short) is shadowed by spam(int)
at example.i:11.
```

To fix this, you either need to ignore or rename one of the methods. For example:

```
%rename(spam_short) spam(short);
...
void spam(int);
void spam(short);    // Accessed as spam_short
```

or

```
%ignore spam(short);
...
void spam(int);
void spam(short);    // Ignored
```

SWIG resolves overloaded functions and methods using a disambiguation scheme that ranks and sorts declarations according to a set of type-precedence rules. The order in which declarations appear in the input does not matter except in situations where ambiguity arises—in this case, the first declaration takes precedence.

Please refer to the "SWIG and C++" chapter for more information about overloading.

### 28.3.11 C++ operators

Certain C++ overloaded operators can be handled automatically by SWIG. For example, consider a class like this:

```
class Complex {
private:
    double rpart, ipart;
public:
    Complex(double r = 0, double i = 0) : rpart(r), ipart(i) { }
    Complex(const Complex &c) : rpart(c.rpart), ipart(c.ipart) { }
    Complex &operator=(const Complex &c);
    Complex operator+(const Complex &c) const;
    Complex operator-(const Complex &c) const;
    Complex operator*(const Complex &c) const;
    Complex operator-() const;

    double re() const { return rpart; }
    double im() const { return ipart; }
};
```

When wrapped, it works like this:

```
% Complex c 3 4
% Complex d 7 8
% set e [c + d]
% $e re
10.0
% $e im
12.0
```

It should be stressed that operators in SWIG have no relationship to operators in Tcl. In fact, the only thing that's happening here is that an operator like `operator +` has been renamed to a method `+`. Therefore, the statement `[c + d]` is really just invoking the `+` method on `c`. When more than operator is defined (with different arguments), the standard method overloading facilities are used. Here is a rather odd looking example:

```
% Complex c 3 4
% Complex d 7 8
% set e [c - d]          # operator-(const Complex &)
% puts "[$e re] [$e im]"
```

```

10.0 12.0
% set f [c -]          # operator-()
% puts "[$f re] [$f im]"
-3.0 -4.0
%

```

One restriction with operator overloading support is that SWIG is not able to fully handle operators that aren't defined as part of the class. For example, if you had code like this

```

class Complex {
...
friend Complex operator+(double, const Complex &c);
...
};

```

then SWIG doesn't know what to do with the friend function—in fact, it simply ignores it and issues a warning. You can still wrap the operator, but you may have to encapsulate it in a special function. For example:

```

%rename(Complex_add_dc) operator+(double, const Complex &);
...
Complex operator+(double, const Complex &c);

```

There are ways to make this operator appear as part of the class using the `%extend` directive. Keep reading.

### 28.3.12 C++ namespaces

SWIG is aware of C++ namespaces, but namespace names do not appear in the module nor do namespaces result in a module that is broken up into submodules or packages. For example, if you have a file like this,

```

%module example

namespace foo {
    int fact(int n);
    struct Vector {
        double x,y,z;
    };
};

```

it works in Tcl as follows:

```

% load ./example.so
% fact 3
6
% Vector v
% v configure -x 3.4

```

If your program has more than one namespace, name conflicts (if any) can be resolved using `%rename`. For example:

```

%rename(Bar_spam) Bar::spam;

namespace Foo {
    int spam();
}

namespace Bar {
    int spam();
}

```

If you have more than one namespace and you want to keep their symbols separate, consider wrapping them as separate SWIG modules. For example, make the module name the same as the namespace and create extension modules for each namespace separately. If your program utilizes thousands of small deeply nested namespaces each with identical symbol names, well, then you get what you deserve.

### 28.3.13 C++ templates

C++ templates don't present a huge problem for SWIG. However, in order to create wrappers, you have to tell SWIG to create wrappers for a particular template instantiation. To do this, you use the `%template` directive. For example:

```
%module example
%{
#include "pair.h"
%}

template<class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1&, const T2&);
    ~pair();
};

%template(pairii) pair<int,int>;
```

In Tcl:

```
% pairii p 3 4
% p cget -first
3
% p cget -second
4
```

Obviously, there is more to template wrapping than shown in this example. More details can be found in the [SWIG and C++](#) chapter. Some more complicated examples will appear later.

### 28.3.14 C++ Smart Pointers

In certain C++ programs, it is common to use classes that have been wrapped by so-called "smart pointers." Generally, this involves the use of a template class that implements `operator->()` like this:

```
template<class T> class SmartPtr {
    ...
    T *operator->();
    ...
}
```

Then, if you have a class like this,

```
class Foo {
public:
    int x;
    int bar();
};
```

A smart pointer would be used in C++ as follows:

```
SmartPtr<Foo> p = CreateFoo();    // Created somehow (not shown)
...
p->x = 3;                        // Foo::x
int y = p->bar();                 // Foo::bar
```

To wrap this in Tcl, simply tell SWIG about the `SmartPtr` class and the low-level `Foo` object. Make sure you instantiate `SmartPtr` using `%template` if necessary. For example:

```
%module example
...
%template(SmartPtrFoo) SmartPtr<Foo>;
...
```

Now, in Tcl, everything should just "work":

```
% set p [CreateFoo]           # Create a smart-pointer somehow
% $p configure -x 3           # Foo::x
% $p bar                      # Foo::bar
```

If you ever need to access the underlying pointer returned by `operator->()` itself, simply use the `__deref__()` method. For example:

```
% set f [$p __deref__]      # Returns underlying Foo *
```

## 28.4 Further details on the Tcl class interface

In the previous section, a high-level view of Tcl wrapping was presented. A key component of this wrapping is that structures and classes are wrapped by Tcl class-like objects. This provides a very natural Tcl interface and allows SWIG to support a number of advanced features such as operator overloading. However, a number of low-level details were omitted. This section provides a brief overview of how the proxy classes work.

### 28.4.1 Proxy classes

In the ["SWIG basics"](#) and ["SWIG and C++"](#) chapters, details of low-level structure and class wrapping are described. To summarize those chapters, if you have a class like this

```
class Foo {
public:
    int x;
    int spam(int);
    ...
}
```

then SWIG transforms it into a set of low-level procedural wrappers. For example:

```
Foo *new_Foo() {
    return new Foo();
}
void delete_Foo(Foo *f) {
    delete f;
}
int Foo_x_get(Foo *f) {
    return f->x;
}
void Foo_x_set(Foo *f, int value) {
    f->x = value;
}
int Foo_spam(Foo *f, int arg1) {
    return f->spam(arg1);
}
```

These wrappers are actually found in the Tcl extension module. For example, you can certainly do this:

```
% load ./example.so
% set f [new_Foo]
% Foo_x_get $f
0
% Foo_x_set $f 3
1
%
```

However, in addition to this, the classname `Foo` is used as an object constructor function. This allows objects to be encapsulated objects that look a lot like Tk widgets as shown in the last section.

## 28.4.2 Memory management

Associated with each wrapped object, is an ownership flag `thisown`. The value of this flag determines who is responsible for deleting the underlying C++ object. If set to 1, the Tcl interpreter destroys the C++ object when the proxy class is garbage collected. If set to 0 (or if the attribute is missing), then the destruction of the proxy class has no effect on the C++ object.

When an object is created by a constructor or returned by value, Tcl automatically takes ownership of the result. For example:

```
class Foo {
public:
    Foo();
    Foo bar();
};
```

In Tcl:

```
% Foo f
% f cget -thisown
1
% set g [f bar]
% $g cget -thisown
1
```

On the other hand, when pointers are returned to Tcl, there is often no way to know where they came from. Therefore, the ownership is set to zero. For example:

```
class Foo {
public:
    ...
    Foo *spam();
    ...
};

% Foo f
% set s [f spam]
% $s cget -thisown
0
%
```

This behavior is especially important for classes that act as containers. For example, if a method returns a pointer to an object that is contained inside another object, you definitely don't want Tcl to assume ownership and destroy it!

Related to containers, ownership issues can arise whenever an object is assigned to a member or global variable. For example, consider this interface:

```
%module example

struct Foo {
    int value;
    Foo *next;
};

Foo *head = 0;
```

When wrapped in Tcl, careful observation will reveal that ownership changes whenever an object is assigned to a global variable. For example:

```
% Foo f
```

```
% f cget -thisown
1
% set head f
% f cget -thisown
0
```

In this case, C is now holding a reference to the object—you probably don't want Tcl to destroy it. Similarly, this occurs for members. For example:

```
% Foo f
% Foo g
% f cget -thisown
1
% g cget -thisown
1
% f configure -next g
% g cget -thisown
0
%
```

For the most part, memory management issues remain hidden. However, there are occasionally situations where you might have to manually change the ownership of an object. For instance, consider code like this:

```
class Node {
    Object *value;
public:
    void set_value(Object *v) { value = v; }
    ...
};
```

Now, consider the following Tcl code:

```
% Object v          # Create an object
% Node n            # Create a node
% n setvalue v      # Set value
% v cget -thisown
1
%
```

In this case, the object n is holding a reference to v internally. However, SWIG has no way to know that this has occurred. Therefore, Tcl still thinks that it has ownership of the object. Should the proxy object be destroyed, then the C++ destructor will be invoked and n will be holding a stale-pointer. If you're lucky, you will only get a segmentation fault.

To work around this, it is always possible to flip the ownership flag. For example,

```
% v -disown          # Give ownership to C/C++
% v -acquire         # Acquire ownership
```

It is also possible to deal with situations like this using typemaps—an advanced topic discussed later.

## 28.5 Input and output parameters

A common problem in some C programs is handling parameters passed as simple pointers. For example:

```
void add(int x, int y, int *result) {
    *result = x + y;
}
```

or perhaps

```
int sub(int *x, int *y) {
    return *x+*y;
}
```

```
}
```

The easiest way to handle these situations is to use the `typemaps.i` file. For example:

```
%module example
#include "typemaps.i"

void add(int, int, int *OUTPUT);
int sub(int *INPUT, int *INPUT);
```

In Tcl, this allows you to pass simple values instead of pointer. For example:

```
set a [add 3 4]
puts $a
7
```

Notice how the `INPUT` parameters allow integer values to be passed instead of pointers and how the `OUTPUT` parameter creates a return result.

If you don't want to use the names `INPUT` or `OUTPUT`, use the `%apply` directive. For example:

```
%module example
#include "typemaps.i"

%apply int *OUTPUT { int *result };
%apply int *INPUT { int *x, int *y};

void add(int x, int y, int *result);
int sub(int *x, int *y);
```

If a function mutates one of its parameters like this,

```
void negate(int *x) {
    *x = -(*x);
}
```

you can use `INOUT` like this:

```
%include "typemaps.i"
...
void negate(int *INOUT);
```

In Tcl, a mutated parameter shows up as a return value. For example:

```
set a [negate 3]
puts $a
-3
```

The most common use of these special typemap rules is to handle functions that return more than one value. For example, sometimes a function returns a result as well as a special error code:

```
/* send message, return number of bytes sent, along with success code */
int send_message(char *text, int len, int *success);
```

To wrap such a function, simply use the `OUTPUT` rule above. For example:

```
%module example
#include "typemaps.i"
%apply int *OUTPUT { int *success };
...
int send_message(char *text, int *success);
```



When used in Tcl, the function will return multiple values as a list.

```
set r [send_message "Hello World"]
set bytes [lindex $r 0]
set success [lindex $r 1]
```

Another common use of multiple return values are in query functions. For example:

```
void get_dimensions(Matrix *m, int *rows, int *columns);
```

To wrap this, you might use the following:

```
%module example
#include "typemaps.i"
%apply int *OUTPUT { int *rows, int *columns };
...
void get_dimensions(Matrix *m, int *rows, *columns);
```

Now, in Perl:

```
set dim [get_dimensions $m]
set r [lindex $dim 0]
set c [lindex $dim 1]
```

## 28.6 Exception handling

The `%exception` directive can be used to create a user-definable exception handler in charge of converting exceptions in your C/C++ program into Tcl exceptions. The chapter on customization features contains more details, but suppose you extended the array example into a C++ class like the following :

```
class RangeError {}; // Used for an exception

class DoubleArray {
private:
    int n;
    double *ptr;
public:
    // Create a new array of fixed size
    DoubleArray(int size) {
        ptr = new double[size];
        n = size;
    }
    // Destroy an array
    ~DoubleArray() {
        delete ptr;
    }
    // Return the length of the array
    int length() {
        return n;
    }

    // Get an item from the array and perform bounds checking.
    double getitem(int i) {
        if ((i >= 0) && (i < n))
            return ptr[i];
        else
            throw RangeError();
    }

    // Set an item in the array and perform bounds checking.
    void setitem(int i, double val) {
        if ((i >= 0) && (i < n))
            ptr[i] = val;
        else {
```

```

        throw RangeError();
    }
}
};

```

The functions associated with this class can throw a C++ range exception for an out-of-bounds array access. We can catch this in our Tcl extension by specifying the following in an interface file :

```

%exception {
    try {
        $action                // Gets substituted by actual function call
    }
    catch (RangeError) {
        Tcl_SetStringObj(tcl_result, "Array index out-of-bounds");
        return TCL_ERROR;
    }
}

```

As shown, the exception handling code will be added to every wrapper function. Since this is somewhat inefficient. You might consider refining the exception handler to only apply to specific methods like this:

```

%exception getitem {
    try {
        $action
    }
    catch (RangeError) {
        Tcl_SetStringObj(tcl_result, "Array index out-of-bounds");
        return TCL_ERROR;
    }
}

%exception setitem {
    try {
        $action
    }
    catch (RangeError) {
        Tcl_SetStringObj(tcl_result, "Array index out-of-bounds");
        return TCL_ERROR;
    }
}

```

In this case, the exception handler is only attached to methods and functions named `getitem` and `setitem`.

If you had a lot of different methods, you can avoid extra typing by using a macro. For example:

```

#define RANGE_ERROR
{
    try {
        $action
    }
    catch (RangeError) {
        Tcl_SetStringObj(tcl_result, "Array index out-of-bounds");
        return TCL_ERROR;
    }
}
#endif

%exception getitem RANGE_ERROR;
%exception setitem RANGE_ERROR;

```

Since SWIG's exception handling is user-definable, you are not limited to C++ exception handling. See the chapter on ["Customization Features"](#) for more examples.

## 28.7 Typemaps

This section describes how you can modify SWIG's default wrapping behavior for various C/C++ datatypes using the `%typemap` directive. This is an advanced topic that assumes familiarity with the Tcl C API as well as the material in the ["Typemaps"](#) chapter.

Before proceeding, it should be stressed that typemaps are not a required part of using SWIG—the default wrapping behavior is enough in most cases. Typemaps are only used if you want to change some aspect of the primitive C–Tcl interface.

### 28.7.1 What is a typemap?

A typemap is nothing more than a code generation rule that is attached to a specific C datatype. For example, to convert integers from Tcl to C, you might define a typemap like this:

```
%module example

%typemap(in) int {
    if (Tcl_GetIntFromObj(interp,$input,&$1) == TCL_ERROR) return TCL_ERROR;
    printf("Received an integer : %d\n",$1);
}
extern int fact(int n);
```

Typemaps are always associated with some specific aspect of code generation. In this case, the "in" method refers to the conversion of input arguments to C/C++. The datatype `int` is the datatype to which the typemap will be applied. The supplied C code is used to convert values. In this code a number of special variable prefaced by a `$` are used. The `$1` variable is placeholder for a local variable of type `int`. The `$input` variable is the input object of type `Tcl_Obj *`.

When this example is compiled into a Tcl module, it operates as follows:

```
% load ./example.so
% fact 6
Received an integer : 6
720
```

In this example, the typemap is applied to all occurrences of the `int` datatype. You can refine this by supplying an optional parameter name. For example:

```
%module example

%typemap(in) int n {
    if (Tcl_GetIntFromObj(interp,$input,&$1) == TCL_ERROR) return TCL_ERROR;
    printf("n = %d\n",$1);
}
extern int fact(int n);
```

In this case, the typemap code is only attached to arguments that exactly match `int n`.

The application of a typemap to specific datatypes and argument names involves more than simple text-matching—typemaps are fully integrated into the SWIG type-system. When you define a typemap for `int`, that typemap applies to `int` and qualified variations such as `const int`. In addition, the typemap system follows `typedef` declarations. For example:

```
%typemap(in) int n {
    if (Tcl_GetIntFromObj(interp,$input,&$1) == TCL_ERROR) return TCL_ERROR;
    printf("n = %d\n",$1);
}
typedef int Integer;
extern int fact(Integer n);    // Above typemap is applied
```

However, the matching of `typedef` only occurs in one direction. If you defined a typemap for `Integer`, it is not applied to arguments of type `int`.

Typemaps can also be defined for groups of consecutive arguments. For example:

```
%typemap(in) (char *str, int len) {
    $1 = Tcl_GetStringFromObj($input,&$2);
};

int count(char c, char *str, int len);
```

When a multi-argument typemap is defined, the arguments are always handled as a single Tcl object. This allows the function to be used like this (notice how the length parameter is ommitted):

```
% count e "Hello World"
1
```

## 28.7.2 Tcl typemaps

The previous section illustrated an "in" typemap for converting Tcl objects to C. A variety of different typemap methods are defined by the Tcl module. For example, to convert a C integer back into a Tcl object, you might define an "out" typemap like this:

```
%typemap(out) int {
    Tcl_SetObjResult(interp,Tcl_NewIntObj($1));
}
```

The following list details all of the typemap methods that can be used by the Tcl module:

`%typemap(in)`

Converts Tcl objects to input function arguments

`%typemap(out)`

Converts return value of a C function to a Tcl object

`%typemap(varin)`

Assigns a C global variable from a Tcl object

`%typemap(varout)`

Returns a C global variable as a Tcl object

`%typemap(freearg)`

Cleans up a function argument (if necessary)

`%typemap(argout)`

Output argument processing

`%typemap(ret)`

Cleanup of function return values

`%typemap(consttab)`

Creation of Tcl constants (constant table)

`%typemap(constcode)`

Creation of Tcl constants (init function)

`%typemap(memberin)`

Setting of structure/class member data

`%typemap(globalin)`

Setting of C global variables

`%typemap(check)`

Checks function input values.

`%typemap(default)`

Set a default value for an argument (making it optional).

`%typemap(arginit)`

Initialize an argument to a value before any conversions occur.

Examples of these methods will appear shortly.

### 28.7.3 Typemap variables

Within typemap code, a number of special variables prefaced with a \$ may appear. A full list of variables can be found in the ["Typemaps"](#) chapter. This is a list of the most common variables:

`$1`

A C local variable corresponding to the actual type specified in the `%typemap` directive. For input values, this is a C local variable that's supposed to hold an argument value. For output values, this is the raw result that's supposed to be returned to Tcl.

`$input`

A `Tcl_Obj` \* holding a raw Tcl object with an argument or variable value.

`$result`

A `Tcl_Obj` \* that holds the result to be returned to Tcl.

`$1_name`

The parameter name that was matched.

`$1_type`

The actual C datatype matched by the typemap.

`$1_ltype`

An assignable version of the datatype matched by the typemap (a type that can appear on the left-hand-side of a C assignment operation). This type is stripped of qualifiers and may be an altered version of \$1\_type. All arguments and local variables in wrapper functions are declared using this type so that their values can be properly assigned.

\$symname

The Tcl name of the wrapper function being created.

## 28.7.4 Converting a Tcl list to a char \*\*

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Tcl list to be used as a char \*\* object.

```
%module argv

// This tells SWIG to treat char ** as a special case
%typemap(in) char ** {
    Tcl_Obj **listobjv;
    int      nitems;
    int      i;
    if (Tcl_ListObjGetElements(interp, $input, &nitems, &listobjv) == TCL_ERROR) {
        return TCL_ERROR;
    }
    $1 = (char **) malloc((nitems+1)*sizeof(char *));
    for (i = 0; i < nitems; i++) {
        $1[i] = Tcl_GetStringFromObj(listobjv[i],0);
    }
    $1[i] = 0;
}

// This gives SWIG some cleanup code that will get called after the function call
%typemap(freearg) char ** {
    if ($1) {
        free($1);
    }
}

// Now a test functions
%inline %{
int print_args(char **argv) {
    int i = 0;
    while (argv[i]) {
        printf("argv[%d] = %s\n", i,argv[i]);
        i++;
    }
    return i;
}
%}
#include tclsh.i
```

In Tcl:

```
% print_args {John Guido Larry}
argv[0] = John
argv[1] = Guido
argv[2] = Larry
3
```

## 28.7.5 Returning values in arguments

The "argout" typemap can be used to return a value originating from a function argument. For example :

```
// A typemap defining how to return an argument by appending it to the result
%typemap(argout) double *outvalue {
    Tcl_Obj *o = Tcl_NewDoubleObj($1);
    Tcl_ListObjAppendElement(interp,$result,o);
}

// A typemap telling SWIG to ignore an argument for input
// However, we still need to pass a pointer to the C function
%typemap(in,numinputs=0) double *outvalue (double temp) {
    $1 = &temp;
}

// Now a function returning two values
int mypow(double a, double b, double *outvalue) {
    if ((a < 0) || (b < 0)) return -1;
    *outvalue = pow(a,b);
    return 0;
};
```

When wrapped, SWIG matches the argout typemap to the "double \*outvalue" argument. The numinputs=0 specification tells SWIG to simply ignore this argument when generating wrapper code. As a result, a Tcl function using these typemaps will work like this :

```
% mypow 2 3      # Returns two values, a status value and the result
0 8
%
```

## 28.7.6 Useful functions

The following tables provide some functions that may be useful in writing Tcl typemaps.

### Integers

```
Tcl_Obj  *Tcl_NewIntObj(int Value);
void      Tcl_SetIntObj(Tcl_Obj *obj, int Value);
int       Tcl_GetIntFromObj(Tcl_Interp *, Tcl_Obj *obj, int *ip);
```

### Floating Point

```
Tcl_Obj  *Tcl_NewDoubleObj(double Value);
void      Tcl_SetDoubleObj(Tcl_Obj *obj, double value);
int       Tcl_GetDoubleFromObj(Tcl_Interp *, Tcl_Obj *o, double *dp);
```

### Strings

```
Tcl_Obj  *Tcl_NewStringObj(char *str, int len);
void      Tcl_SetStringObj(Tcl_Obj *obj, char *str, int len);
char      *Tcl_GetStringFromObj(Tcl_Obj *obj, int *len);
void      Tcl_AppendToObj(Tcl_Obj *obj, char *str, int len);
```

### Lists

```
Tcl_Obj  *Tcl_NewListObj(int objc, Tcl_Obj *objv);
int       Tcl_ListObjAppendList(Tcl_Interp *, Tcl_Obj *listPtr, Tcl_Obj *elemListPtr);
int       Tcl_ListObjAppendElement(Tcl_Interp *, Tcl_Obj *listPtr, Tcl_Obj *element);
int       Tcl_ListObjGetElements(Tcl_Interp *, Tcl_Obj *listPtr, int *objcPtr,
                                Tcl_Obj ***objvPtr);
int       Tcl_ListObjLength(Tcl_Interp *, Tcl_Obj *listPtr, int *intPtr);
```

```

int      Tcl_ListObjIndex(Tcl_Interp *, Tcl_Obj *listPtr, int index,
                        Tcl_Obj_Obj **objPtr);
int      Tcl_ListObjReplace(Tcl_Interp *, Tcl_Obj *listPtr, int first, int count,
                        int objc, Tcl_Obj *objv);

```

## Objects

```

Tcl_Obj *Tcl_DuplicateObj(Tcl_Obj *obj);
void      Tcl_IncrRefCount(Tcl_Obj *obj);
void      Tcl_DecrRefCount(Tcl_Obj *obj);
int      Tcl_IsShared(Tcl_Obj *obj);

```

## 28.7.7 Standard typemaps

The following typemaps show how to convert a few common kinds of objects between Tcl and C (and to give a better idea of how typemaps work)

### Integer conversion

```

%typemap(in) int, short, long {
    int temp;
    if (Tcl_GetIntFromObj(interp, $input, &temp) == TCL_ERROR)
        return TCL_ERROR;
    $l = ($l_ltype) temp;
}

%typemap(out) int, short, long {
    Tcl_SetIntObj($result, (int) $l);
}

```

### Floating point conversion

```

%typemap(in) float, double {
    double temp;
    if (Tcl_GetDoubleFromObj(interp, $input, &temp) == TCL_ERROR)
        return TCL_ERROR;
    $l = ($l_ltype) temp;
}

%typemap(out) float, double {
    Tcl_SetDoubleObj($result, $l);
}

```

### String Conversion

```

%typemap(in) char * {
    int len;
    $l = Tcl_GetStringFromObj(interp, &len);
}

%typemap(out) char * {
    Tcl_SetStringObj($result, $l);
}

```

## 28.7.8 Pointer handling

SWIG pointers are mapped into Tcl strings containing the hexadecimal value and type. The following functions can be used to create and read pointer values.

```

int SWIG_ConvertPtr(Tcl_Obj *obj, void **ptr, swig_type_info *ty, int flags)

```



## SWIG-1.3 Documentation

Converts a Tcl object `obj` to a C pointer. The result of the conversion is placed into the pointer located at `ptr`. `ty` is a SWIG type descriptor structure. `flags` is used to handle error checking and other aspects of conversion. It is currently reserved for future expansion. Returns 0 on success and -1 on error.

```
Tcl_Obj *SWIG_NewPointerObj(void *ptr, swig_type_info *ty, int flags)
```

Creates a new Tcl pointer object. `ptr` is the pointer to convert, `ty` is the SWIG type descriptor structure that describes the type, and `own` is a flag reserved for future expansion.

Both of these functions require the use of a special SWIG type-descriptor structure. This structure contains information about the mangled name of the datatype, type-equivalence information, as well as information about converting pointer values under C++ inheritance. For a type of `Foo *`, the type descriptor structure is usually accessed as follows:

```
Foo *f;
if (SWIG_ConvertPtr($input, (void **) &f, SWIGTYPE_p_Foo, 0) == -1) return NULL;

Tcl_Obj *;
obj = SWIG_NewPointerObj(f, SWIGTYPE_p_Foo, 0);
```

In a typemap, the type descriptor should always be accessed using the special typemap variable `$l_descriptor`. For example:

```
%typemap(in) Foo * {
    if ((SWIG_ConvertPtr($input, (void **) &$l, $l_descriptor, 0)) == -1) return NULL;
}
```

If necessary, the descriptor for any type can be obtained using the `$descriptor()` macro in a typemap. For example:

```
%typemap(in) Foo * {
    if ((SWIG_ConvertPtr($input, (void **) &$l, $descriptor(Foo *), 0)) == -1) return NULL;
}
```

## 28.8 Turning a SWIG module into a Tcl Package.

Tcl 7.4 introduced the idea of an extension package. By default, SWIG generates all of the code necessary to create a package. To set the package version, simply use the `-pkgversion` option. For example:

```
% swig -tcl -pkgversion 2.3 example.i
```

After building the SWIG generated module, you need to execute the "pkg\_mkIndex" command inside tclsh. For example :

```
unix > tclsh
% pkg_mkIndex . example.so
% exit
```

This creates a file "pkgIndex.tcl" with information about the package. To use your package, you now need to move it to its own subdirectory which has the same name as the package. For example :

```
./example/
    pkgIndex.tcl      # The file created by pkg_mkIndex
    example.so        # The SWIG generated module
```

Finally, assuming that you're not entirely confused at this point, make sure that the example subdirectory is visible from the directories contained in either the `tcl_library` or `auto_path` variables. At this point you're ready to use the package as follows :

```
unix > tclsh
% package require example
% fact 4
24
%
```

If you're working with an example in the current directory and this doesn't work, do this instead :

```
unix > tclsh
% lappend auto_path .
% package require example
% fact 4
24
```

As a final note, most SWIG examples do not yet use the package commands. For simple extensions it may be easier just to use the load command instead.

## 28.9 Building new kinds of Tcl interfaces (in Tcl)

One of the most interesting aspects of Tcl and SWIG is that you can create entirely new kinds of Tcl interfaces in Tcl using the low-level SWIG accessor functions. For example, suppose you had a library of helper functions to access arrays :

```
/* File : array.i */
%module array

%inline %{
double *new_double(int size) {
    return (double *) malloc(size*sizeof(double));
}
void delete_double(double *a) {
    free(a);
}
double get_double(double *a, int index) {
    return a[index];
}
void set_double(double *a, int index, double val) {
    a[index] = val;
}
int *new_int(int size) {
    return (int *) malloc(size*sizeof(int));
}
void delete_int(int *a) {
    free(a);
}
int get_int(int *a, int index) {
    return a[index];
}
int set_int(int *a, int index, int val) {
    a[index] = val;
}
%}
```

While these could be called directly, we could also write a Tcl script like this :

```
proc Array {type size} {
    set ptr [new_$type $size]
    set code {
        set method [lindex $args 0]
        set parms [concat $ptr [lrange $args 1 end]]
        switch $method {
            get {return [eval "get_$type $parms"]}
            set {return [eval "set_$type $parms"]}
            delete {eval "delete_$type $ptr; rename $ptr {}"}
        }
    }
    # Create a procedure
    uplevel "proc $ptr args {set ptr $ptr; set type $type;$code}"
    return $ptr
}
```

Our script allows easy array access as follows :

```

set a [Array double 100]           ;# Create a double [100]
for {set i 0} {$i < 100} {incr i 1} { ;# Clear the array
    $a set $i 0.0
}
$a set 3 3.1455                    ;# Set an individual element
set b [$a get 10]                  ;# Retrieve an element

set ia [Array int 50]              ;# Create an int[50]
for {set i 0} {$i < 50} {incr i 1} { ;# Clear it
    $ia set $i 0
}
$ia set 3 7                        ;# Set an individual element
set ib [$ia get 10]                 ;# Get an individual element

$a delete                          ;# Destroy a
$ia delete                          ;# Destroy ia

```

The cool thing about this approach is that it makes a common interface for two different types of arrays. In fact, if we were to add more C datatypes to our wrapper file, the Tcl code would work with those as well—without modification. If an unsupported datatype was requested, the Tcl code would simply return with an error so there is very little danger of blowing something up (although it is easily accomplished with an out of bounds array access).

### 28.9.1 Proxy classes

A similar approach can be applied to proxy classes (also known as shadow classes). The following example is provided by Erik Bierwagen and Paul Saxe. To use it, run SWIG with the `-noobject` option (which disables the builtin object oriented interface). When running Tcl, simply source this file. Now, objects can be used in a more or less natural fashion.

```

# swig_c++.tcl
# Provides a simple object oriented interface using
# SWIG's low level interface.
#

proc new {objectType handle_r args} {
    # Creates a new SWIG object of the given type,
    # returning a handle in the variable "handle_r".
    #
    # Also creates a procedure for the object and a trace on
    # the handle variable that deletes the object when the
    # handle variable is overwritten or unset
    upvar $handle_r handle
    #
    # Create the new object
    #
    eval set handle \[new_$objectType $args\]
    #
    # Set up the object procedure
    #
    proc $handle {cmd args} "eval ${objectType}_\${cmd} $handle $args"
    #
    # And the trace ...
    #
    uplevel trace variable $handle_r uw "{deleteObject $objectType $handle}"
    #
    # Return the handle so that 'new' can be used as an argument to a procedure
    #
    return $handle
}

proc deleteObject {objectType handle name element op} {
    #
    # Check that the object handle has a reasonable form
    #

```

## SWIG-1.3 Documentation

```
if {[regexp {[0-9a-f]*_(.)_p} $handle]} {
    error "deleteObject: not a valid object handle: $handle"
}
#
# Remove the object procedure
#
catch {rename $handle {}}
#
# Delete the object
#
delete_$objectType $handle
}

proc delete {handle_r} {
    #
    # A synonym for unset that is more familiar to C++ programmers
    #
    uplevel unset $handle_r
}
```

To use this file, we simply source it and execute commands such as "new" and "delete" to manipulate objects. For example :

```
// list.i
%module List
%{
#include "list.h"
%}

// Very simple C++ example

class List {
public:
    List(); // Create a new list
    ~List(); // Destroy a list
    int search(char *value);
    void insert(char *); // Insert a new item into the list
    void remove(char *); // Remove item from list
    char *get(int n); // Get the nth item in the list
    int length; // The current length of the list
    static void print(List *l); // Print out the contents of the list
};
```

Now a Tcl script using the interface...

```
load ./list.so list      ; # Load the module
source swig_c++.tcl      ; # Source the object file

new List l
$l insert Dave
$l insert John
$l insert Guido
$l remove Dave
puts $l length_get

delete l
```

The cool thing about this example is that it works with any C++ object wrapped by SWIG and requires no special compilation. Proof that a short, but clever Tcl script can be combined with SWIG to do many interesting things.

## 29 Extending SWIG

- [Introduction](#)
- [Prerequisites](#)
- [The Big Picture](#)
- [Execution Model](#)
  - ◆ [Preprocessing](#)
  - ◆ [Parsing](#)
  - ◆ [Parse Trees](#)
  - ◆ [Attribute namespaces](#)
  - ◆ [Symbol Tables](#)
  - ◆ [The %feature directive](#)
  - ◆ [Code Generation](#)
  - ◆ [SWIG and XML](#)
- [Primitive Data Structures](#)
  - ◆ [Strings](#)
  - ◆ [Hashes](#)
  - ◆ [Lists](#)
  - ◆ [Common operations](#)
  - ◆ [Iterating over Lists and Hashes](#)
  - ◆ [I/O](#)
- [Navigating and manipulating parse trees](#)
- [Working with attributes](#)
- [Type system](#)
  - ◆ [String encoding of types](#)
  - ◆ [Type construction](#)
  - ◆ [Type tests](#)
  - ◆ [Typedef and inheritance](#)
  - ◆ [Lvalues](#)
  - ◆ [Output functions](#)
- [Parameters](#)
- [Writing a Language Module](#)
  - ◆ [Execution model](#)
  - ◆ [Starting out](#)
  - ◆ [Command line options](#)
  - ◆ [Configuration and preprocessing](#)
  - ◆ [Entry point to code generation](#)
  - ◆ [Module I/O and wrapper skeleton](#)
  - ◆ [Low-level code generators](#)
  - ◆ [Configuration files](#)
  - ◆ [Runtime support](#)
  - ◆ [Standard library files](#)
  - ◆ [Examples and test cases](#)
  - ◆ [Documentation](#)
- [Typemaps](#)
  - ◆ [Proxy classes](#)
- [Guide to parse tree nodes](#)

**Caution:** This chapter is being rewritten! (11/25/01)

### 29.1 Introduction

This chapter describes SWIG's internal organization and the process by which new target languages can be developed. First, a brief word of warning——SWIG has been undergoing a massive redevelopment effort that has focused extensively on its internal organization. The information in this chapter is mostly up to date, but changes are ongoing. Expect a few inconsistencies.

Also, this chapter is not meant to be a hand-holding tutorial. As a starting point, you should probably look at one of SWIG's existing modules.

## 29.2 Prerequisites

In order to extend SWIG, it is useful to have the following background:

- An understanding of the C API for the target language.
- A good grasp of the C++ type system.
- An understanding of typemaps and some of SWIG's advanced features.
- Some familiarity with writing C++ (language modules are currently written in C++).

Since SWIG is essentially a specialized C++ compiler, it may be useful to have some prior experience with compiler design (perhaps even a compilers course) to better understand certain parts of the system. A number of books will also be useful. For example, "The C Programming Language" by Kernighan and Ritchie (a.k.a, "K&R") and the "C++ Annotated Reference Manual" by Stroustrup (a.k.a, the "ARM") will be of great use.

Also, it is useful to keep in mind that SWIG primarily operates as an extension of the C++ *type* system. At first glance, this might not be obvious, but almost all SWIG directives as well as the low-level generation of wrapper code are driven by C++ datatypes.

## 29.3 The Big Picture

SWIG is a special purpose compiler that parses C++ declarations to generate wrapper code. To make this conversion possible, SWIG makes three fundamental extensions to the C++ language:

- **Typemaps.** Typemaps are used to define the conversion/marshalling behavior of specific C++ datatypes. All type conversion in SWIG is based on typemaps. Furthermore, the association of typemaps to datatypes utilizes an advanced pattern matching mechanism that is fully integrated with the C++ type system.
- **Declaration Annotation.** To customize wrapper code generation, most declarations can be annotated with special features. For example, you can make a variable read-only, you can ignore a declaration, you can rename a member function, you can add exception handling, and so forth. Virtually all of these customizations are built on top of a low-level declaration annotator that can attach arbitrary attributes to any declaration. Code generation modules can look for these attributes to guide the wrapping process.
- **Class extension.** SWIG allows classes and structures to be extended with new methods and attributes (the `%extend` directive). This has the effect of altering the API in the target language and can be used to generate OO interfaces to C libraries.

It is important to emphasize that virtually all SWIG features reduce to one of these three fundamental concepts. The type system and pattern matching rules also play a critical role in making the system work. For example, both typemaps and declaration annotation are based on pattern matching and interact heavily with the underlying type system.

## 29.4 Execution Model

When you run SWIG on an interface, processing is handled in stages by a series of system components:

- An integrated C preprocessor reads a collection of configuration files and the specified interface file into memory. The preprocessor performs the usual functions including macro expansion and file inclusion. However, the preprocessor also performs some transformations of the interface. For instance, `#define` statements are sometimes transformed into `%constant` declarations. In addition, information related to file/line number tracking is inserted.
- A C/C++ parser reads the preprocessed input and generates a full parse tree of all of the SWIG directives and C declarations found. The parser is responsible for many aspects of the system including renaming, declaration annotation, and template expansion. However, the parser does not produce any output nor does it interact with the target language module as it runs. SWIG is not a one-pass compiler.
- A type-checking pass is made. This adjusts all of the C++ typenames to properly handle namespaces, typedefs, nested classes, and other issues related to type scoping.

- A semantic pass is made on the parse tree to collect information related to properties of the C++ interface. For example, this pass would determine whether or not a class allows a default constructor.
- A code generation pass is made using a specific target language module. This phase is responsible for generating the actual wrapper code. All of SWIG's user-defined modules are invoked during this stage of compilation.

The next few sections briefly describe some of these stages.

## 29.4.1 Preprocessing

The preprocessor plays a critical role in the SWIG implementation. This is because a lot of SWIG's processing and internal configuration is managed not by code written in C, but by configuration files in the SWIG library. In fact, when you run SWIG, parsing starts with a small interface file like this (note: this explains the cryptic error messages that new users sometimes get when SWIG is misconfigured or installed incorrectly):

```
%include "swig.swg"           // Global SWIG configuration
%include "langconfig.swg"     // Language specific configuration
%include "yourinterface.i"    // Your interface file
```

The `swig.swg` file contains global configuration information. In addition, this file defines many of SWIG's standard directives as macros. For instance, part of `swig.swg` looks like this:

```
...
/* Code insertion directives such as %wrapper %{ ... %} */

#define %init          %insert("init")
#define %wrapper       %insert("wrapper")
#define %header        %insert("header")
#define %runtime       %insert("runtime")

/* Access control directives */

#define %immutable     %feature("immutable","1")
#define %mutable      %feature("immutable")

/* Directives for callback functions */

#define %callback(x) %feature("callback") `x`;
#define %nocallback  %feature("callback");

/* %ignore directive */

#define %ignore        %rename($ignore)
#define %ignorewarn(x) %rename("$ignore:" x)
...
```

The fact that most of the standard SWIG directives are macros is intended to simplify the implementation of the internals. For instance, rather than having to support dozens of special directives, it is easier to have a few basic primitives such as `%feature` or `%insert`.

The `langconfig.swg` file is supplied by the target language. This file contains language-specific configuration information. More often than not, this file provides run-time wrapper support code (e.g., the type-checker) as well as a collection of typemaps that define the default wrapping behavior. Note: the name of this file depends on the target language and is usually something like `python.swg` or `perl5.swg`.

As a debugging aide, the text that SWIG feeds to its C++ parser can be obtained by running `swig -E interface.i`. This output probably isn't too useful in general, but it will show how macros have been expanded as well as everything else that goes into the low-level construction of the wrapper code.

## 29.4.2 Parsing

The current C++ parser handles a subset of C++. Most incompatibilities with C are due to subtle aspects of how SWIG parses declarations. Specifically, SWIG expects all C/C++ declarations to follow this general form:

```
storage type declarator initializer;
```

*storage* is a keyword such as `extern`, `static`, `typedef`, or `virtual`. *type* is a primitive datatype such as `int` or `void`. *type* may be optionally qualified with a qualifier such as `const` or `volatile`. *declarator* is a name with additional type-construction modifiers attached to it (pointers, arrays, references, functions, etc.). Examples of declarators include `*x`, `**x`, `x[20]`, and `(*x)(int, double)`. The *initializer* may be a value assigned using `=` or body of code enclosed in braces `{ ... }`.

This declaration format covers most common C++ declarations. However, the C++ standard is somewhat more flexible in the placement of the parts. For example, it is technically legal, although uncommon to write something like `int typedef const` in your program. SWIG simply doesn't bother to deal with this case.

The other significant difference between C++ and SWIG is in the treatment of typenames. In C++, if you have a declaration like this,

```
int blah(Foo *x, Bar *y);
```

it won't parse correctly unless `Foo` and `Bar` have been previously defined as types either using a `class` definition or a `typedef`. The reasons for this are subtle, but this treatment of typenames is normally integrated at the level of the C tokenizer—when a typename appears, a different token is returned to the parser instead of an identifier.

SWIG does not operate in this manner—any legal identifier can be used as a type name. The reason for this is primarily motivated by the use of SWIG with partially defined data. Specifically, SWIG is supposed to be easy to use on interfaces with missing type information.

Because of the different treatment of typenames, the most serious limitation of the SWIG parser is that it can't process type declarations where an extra (and unnecessary) grouping operator is used. For example:

```
int (x);           /* A variable x */
int (y)(int);      /* A function y */
```

The placing of extra parentheses in type declarations like this is already recognized by the C++ community as a potential source of strange programming errors. For example, Scott Meyers "Effective STL" discusses this problem in a section on avoiding C++'s "most vexing parse."

The parser is also unable to handle declarations with no return type or bare argument names. For example, in an old C program, you might see things like this:

```
foo(a,b) {
...
}
```

In this case, the return type as well as the types of the arguments are taken by the C compiler to be an `int`. However, SWIG interprets the above code as an abstract declarator for a function returning a `foo` and taking types `a` and `b` as arguments).

## 29.4.3 Parse Trees

The SWIG parser produces a complete parse tree of the input file before any wrapper code is actually generated. Each item in the tree is known as a "Node". Each node is identified by a symbolic tag. Furthermore, a node may have an arbitrary number of children. The parse tree structure and tag names of an interface can be displayed using `swig -dump_tags example.i`. For example:

```
$ swig -c++ -python -dump_tags example.i
```



## SWIG-1.3 Documentation

```
. top (example.i:1)
. top . include (example.i:1)
. top . include . typemap (/r0/beazley/Projects/lib/swigl.3/swig.swg:71)
. top . include . typemap . typemapitem (/r0/beazley/Projects/lib/swigl.3/swig.swg:71)
. top . include . typemap (/r0/beazley/Projects/lib/swigl.3/swig.swg:83)
. top . include . typemap . typemapitem (/r0/beazley/Projects/lib/swigl.3/swig.swg:83)
. top . include (example.i:4)
. top . include . insert (/r0/beazley/Projects/lib/swigl.3/python/python.swg:7)
. top . include . insert (/r0/beazley/Projects/lib/swigl.3/python/python.swg:8)
. top . include . typemap (/r0/beazley/Projects/lib/swigl.3/python/python.swg:19)
...
. top . include (example.i:6)
. top . include . module (example.i:2)
. top . include . insert (example.i:6)
. top . include . include (example.i:9)
. top . include . include . class (example.h:3)
. top . include . include . class . access (example.h:4)
. top . include . include . class . constructor (example.h:7)
. top . include . include . class . destructor (example.h:10)
. top . include . include . class . cdecl (example.h:11)
. top . include . include . class . cdecl (example.h:11)
. top . include . include . class . cdecl (example.h:12)
. top . include . include . class . cdecl (example.h:13)
. top . include . include . class . cdecl (example.h:14)
. top . include . include . class . cdecl (example.h:15)
. top . include . include . class (example.h:18)
. top . include . include . class . access (example.h:19)
. top . include . include . class . cdecl (example.h:20)
. top . include . include . class . access (example.h:21)
. top . include . include . class . constructor (example.h:22)
. top . include . include . class . cdecl (example.h:23)
. top . include . include . class . cdecl (example.h:24)
. top . include . include . include . class (example.h:27)
. top . include . include . class . access (example.h:28)
. top . include . include . class . cdecl (example.h:29)
. top . include . include . class . access (example.h:30)
. top . include . include . class . constructor (example.h:31)
. top . include . include . class . cdecl (example.h:32)
. top . include . include . class . cdecl (example.h:33)
```

Even for the most simple interface, the parse tree structure is larger than you might expect. For example, in the above output, a substantial number of nodes are actually generated by the `python.swg` configuration file which defines typemaps and other directives. The contents of the user-supplied input file don't appear until the end of the output.

The contents of each parse tree node consist of a collection of attribute/value pairs. Internally, the nodes are simply represented by hash tables. A display of the parse-tree structure can be obtained using `swig -dump tree`. For example:

```
$ swig -c++ -python -dump_tree example.i
...
+++ include -----
| name           - "example.i"

+++ module -----
| name           - "example"
|

+++ insert -----
| code           - "\n#include \"example.h\"\n"
|

+++ include -----
| name           - "example.h"

+++ class -----
| abstract       - "1"
| sym:name       - "Shape"
| name           - "Shape"
| kind           - "class"
```

## SWIG-1.3 Documentation

```
| symtab      - 0x40194140
| sym:symtab  - 0x40191078

+++ access -----
| kind        - "public"
|
+++ constructor -----
| sym:name     - "Shape"
| name        - "Shape"
| decl        - "f()."
| code        - "{\n      nshapes++; \n  }"
| sym:symtab  - 0x40194140
|
+++ destructor -----
| sym:name     - "~Shape"
| name        - "~Shape"
| storage     - "virtual"
| code        - "{\n      nshapes--; \n  }"
| sym:symtab  - 0x40194140
|
+++ cdecl -----
| sym:name     - "x"
| name        - "x"
| decl        - ""
| type        - "double"
| sym:symtab  - 0x40194140
|
+++ cdecl -----
| sym:name     - "y"
| name        - "y"
| decl        - ""
| type        - "double"
| sym:symtab  - 0x40194140
|
+++ cdecl -----
| sym:name     - "move"
| name        - "move"
| decl        - "f(double,double)."

```

## SWIG-1.3 Documentation

```

| sym:symtab      - 0x40194140
+++ class -----
| sym:name        - "Circle"
| name            - "Circle"
| kind            - "class"
| bases           - 0x40194510
| symtab          - 0x40194538
| sym:symtab      - 0x40191078
|
| +++ access -----
| kind            - "private"
|
| +++ cdecl -----
| name            - "radius"
| decl            - ""
| type            - "double"
|
| +++ access -----
| kind            - "public"
|
| +++ constructor -----
| sym:name        - "Circle"
| name            - "Circle"
| parms           - double
| decl            - "f(double). "
| code            - "{ }"
| sym:symtab      - 0x40194538
|
| +++ cdecl -----
| sym:name        - "area"
| name            - "area"
| decl            - "f(void). "
| parms           - void
| storage         - "virtual"
| type            - "double"
| sym:symtab      - 0x40194538
|
| +++ cdecl -----
| sym:name        - "perimeter"
| name            - "perimeter"
| decl            - "f(void). "
| parms           - void
| storage         - "virtual"
| type            - "double"
| sym:symtab      - 0x40194538
|
+++ class -----
| sym:name        - "Square"
| name            - "Square"
| kind            - "class"
| bases           - 0x40194760
| symtab          - 0x40194788
| sym:symtab      - 0x40191078
|
| +++ access -----
| kind            - "private"
|
| +++ cdecl -----
| name            - "width"
| decl            - ""
| type            - "double"
|
| +++ access -----
| kind            - "public"
|
| +++ constructor -----
| sym:name        - "Square"

```

```

| name          - "Square"
| parms         - double
| decl          - "f(double).\"
| code          - "{ }"
| sym:symtab    - 0x40194788
|
+++ cdecl -----
| sym:name      - "area"
| name         - "area"
| decl         - "f(void).\"
| parms        - void
| storage      - "virtual"
| type         - "double"
| sym:symtab   - 0x40194788
|
+++ cdecl -----
| sym:name      - "perimeter"
| name         - "perimeter"
| decl         - "f(void).\"
| parms        - void
| storage      - "virtual"
| type         - "double"
| sym:symtab   - 0x40194788

```

### 29.4.4 Attribute namespaces

Attributes of parse tree nodes are often prepended with a namespace qualifier. For example, the attributes `sym:name` and `sym:symtab` are attributes related to symbol table management and are prefixed with `sym:.` As a general rule, only those attributes which are directly related to the raw declaration appear without a prefix (type, name, declarator, etc.).

Target language modules may add additional attributes to nodes to assist the generation of wrapper code. The convention for doing this is to place these attributes in a namespace that matches the name of the target language. For example, `python:foo` or `perl:foo`.

### 29.4.5 Symbol Tables

During parsing, all symbols are managed in the space of the target language. The `sym:name` attribute of each node contains the symbol name selected by the parser. Normally, `sym:name` and `name` are the same. However, the `%rename` directive can be used to change the value of `sym:name`. You can see the effect of `%rename` by trying it on a simple interface and dumping the parse tree. For example:

```

%rename(foo_i) foo(int);
%rename(foo_d) foo(double);

void foo(int);
void foo(double);
void foo(Bar *b);

```

Now, running SWIG:

```

$ swig -dump_tree example.i
...
+++ cdecl -----
| sym:name      - "foo_i"
| name         - "foo"
| decl         - "f(int).\"
| parms        - int
| type         - "void"
| sym:symtab   - 0x40165078
|
+++ cdecl -----
| sym:name      - "foo_d"
| name         - "foo"

```

```

| decl      - "f(double). "
| parms     - double
| type      - "void"
| sym:symtab - 0x40165078
|
+++ cdecl -----
| sym:name   - "foo"
| name       - "foo"
| decl       - "f(p.Bar). "
| parms     - Bar *
| type       - "void"
| sym:symtab - 0x40165078

```

All symbol-related conflicts and complaints about overloading are based on `sym:name` values. For instance, the following example uses `%rename` in reverse to generate a name clash.

```

%rename(foo) foo_i(int);
%rename(foo) foo_d(double);

void foo_i(int);
void foo_d(double);
void foo(Bar *b);

```

When you run SWIG on this you now get:

```

$ ./swig example.i
example.i:6. Overloaded declaration ignored.  foo_d(double )
example.i:5. Previous declaration is foo_i(int )
example.i:7. Overloaded declaration ignored.  foo(Bar *)
example.i:5. Previous declaration is foo_i(int )

```

## 29.4.6 The %feature directive

A number of SWIG directives such as `%exception` are implemented using the low-level `%feature` directive. For example:

```

%feature("except") getitem(int) {
    try {
        $action
    } catch (badindex) {
        ...
    }
}

...
class Foo {
public:
    Object *getitem(int index) throws(badindex);
    ...
};

```

The behavior of `%feature` is very easy to describe—it simply attaches a new attribute to any parse tree node that matches the given prototype. When a feature is added, it shows up as an attribute in the `feature:` namespace. You can see this when running with the `-dump_tree` option. For example:

```

+++ cdecl -----
| sym:name     - "getitem"
| name         - "getitem"
| decl         - "f(int).p."
| parms       - int
| type         - "Object"
| feature:except - "{\n    try {\n        $action\n    } catc..."
| sym:symtab   - 0x40168ac8
|

```

Feature names are completely arbitrary and a target language module can be programmed to respond to any feature name that it wants to be recognized. The data stored in a feature attribute is usually just a raw unparsed string. For example, the exception code above is simply stored without any modifications.

## 29.4.7 Code Generation

Language modules work by defining handler functions that know how to respond to different types of parse-tree nodes. These handlers simply look at the attributes of each node in order to produce low-level code.

In reality, the generation of code is somewhat more subtle than simply invoking handler functions. This is because parse-tree nodes might be transformed. For example, suppose you are wrapping a class like this:

```
class Foo {
public:
    virtual int *bar(int x);
};
```

When the parser constructs a node for the member `bar`, it creates a raw "cdecl" node with the following attributes:

```
nodeType      : cdecl
name          : bar
type          : int
decl          : f(int).p
parms         : int x
storage       : virtual
sym:name      : bar
```

To produce wrapper code, this "cdecl" node undergoes a number of transformations. First, the node is recognized as a function declaration. This adjusts some of the type information—specifically, the declarator is joined with the base datatype to produce this:

```
nodeType      : cdecl
name          : bar
type          : p.int      <-- Notice change in return type
decl          : f(int).p
parms         : int x
storage       : virtual
sym:name      : bar
```

Next, the context of the node indicates that the node is really a member function. This produces a transformation to a low-level accessor function like this:

```
nodeType      : cdecl
name          : bar
type          : int.p
decl          : f(int).p
parms         : Foo *self, int x      <-- Added parameter
storage       : virtual
wrap:action   : result = (arg1)->bar(arg2) <-- Action code added
sym:name      : Foo_bar               <-- Symbol name changed
```

In this transformation, notice how an additional parameter was added to the parameter list and how the symbol name of the node has suddenly changed into an accessor using the naming scheme described in the "SWIG Basics" chapter. A small fragment of "action" code has also been generated—notice how the `wrap:action` attribute defines the access to the underlying method. The data in this transformed node is then used to generate a wrapper.

Language modules work by registering handler functions for dealing with various types of nodes at different stages of transformation. This is done by inheriting from a special `Language` class and defining a collection of virtual methods. For example, the Python module defines a class as follows:

```
class PYTHON : public Language {
```

```
protected:
public :
    virtual void main(int, char *argv[]);
    virtual int  top(Node *);
    virtual int  functionWrapper(Node *);
    virtual int  constantWrapper(Node *);
    virtual int  variableWrapper(Node *);
    virtual int  nativeWrapper(Node *);
    virtual int  membervariableHandler(Node *);
    virtual int  memberconstantHandler(Node *);
    virtual int  memberfunctionHandler(Node *);
    virtual int  constructorHandler(Node *);
    virtual int  destructorHandler(Node *);
    virtual int  classHandler(Node *);
    virtual int  classforwardDeclaration(Node *);
    virtual int  insertDirective(Node *);
    virtual int  importDirective(Node *);
};
```

The role of these functions is described shortly.

## 29.4.8 SWIG and XML

Much of SWIG's current parser design was originally motivated by interest in using XML to represent SWIG parse trees. Although XML is not currently used in any direct manner, the parse tree structure, use of node tags, attributes, and attribute namespaces are all influenced by aspects of XML parsing. Therefore, in trying to understand SWIG's internal data structures, it may be useful keep XML in the back of your mind as a model.

## 29.5 Primitive Data Structures

Most of SWIG is constructed using three basic data structures: strings, hashes, and lists. These data structures are dynamic in same way as similar structures found in many scripting languages. For instance, you can have containers (lists and hash tables) of mixed types and certain operations are polymorphic.

This section briefly describes the basic structures so that later sections of this chapter make more sense.

When describing the low-level API, the following type name conventions are used:

- **String**. A string object.
- **Hash**. A hash object.
- **List**. A list object.
- **String\_or\_char**. A string object or a `char *`.
- **Object\_or\_char**. An object or a `char *`.
- **Object**. Any object (string, hash, list, etc.)

In most cases, other typenames in the source are aliases for one of these primitive types. Specifically:

```
typedef String SwigType;
typedef Hash  Parm;
typedef Hash  ParmList;
typedef Hash  Node;
typedef Hash  Symtab;
typedef Hash  Typetab;
```

### 29.5.1 Strings

```
String *NewString(const String_or_char *val)
```

Creates a new string with initial value `val`. `val` may be a `char *` or another **String** object. If you want to

create an empty string, use "" for val.

**String \*NewStringf(const char \*fmt, ...)**

Creates a new string whose initial value is set according to a C printf style format string in *fmt*. Additional arguments follow depending on *fmt*.

**String \*Copy(String \*s)**

Make a copy of the string *s*.

**void Delete(String \*s)**

Deletes *s*.

**int Len(String\_or\_char \*s)**

Returns the length of the string.

**char \*Char(String\_or\_char \*s)**

Returns a pointer to the first character in a string.

**void Append(String \*s, String\_or\_char \*t)**

Appends *t* to the end of string *s*.

**void Insert(String \*s, int pos, String\_or\_char \*t)**

Inserts *t* into *s* at position *pos*. The contents of *s* are shifted accordingly. The special value DOH\_END can be used for *pos* to indicate insertion at the end of the string (appending).

**int Strcmp(const String\_or\_char \*s, const String\_or\_char \*t)**

Compare strings *s* and *t*. Same as the C `strcmp()` function.

**int Strncmp(const String\_or\_char \*s, const String\_or\_char \*t, int len)**

Compare the first *len* characters of strings *s* and *t*. Same as the C `strncmp()` function.

**char \*Strstr(const String\_or\_char \*s, const String\_or\_char \*pat)**

Returns a pointer to the first occurrence of *pat* in *s*. Same as the C `strstr()` function.

**char \*Strchr(const String\_or\_char \*s, char ch)**

Returns a pointer to the first occurrence of character *ch* in *s*. Same as the C `strchr()` function.

**void Chop(String \*s)**

Chops trailing whitespace off the end of *s*.

**int Replace(String \*s, const String\_or\_char \*pat, const String\_or\_char \*rep, int flags)**

Replaces the pattern *pat* with *rep* in string *s*. *flags* is a combination of the following flags:



DOH_REPLACE_ANY	- Replace all occurrences
DOH_REPLACE_ID	- Valid C identifiers only
DOH_REPLACE_NOQUOTE	- Don't replace in quoted strings
DOH_REPLACE_FIRST	- Replace first occurrence only.

Returns the number of replacements made (if any).

## 29.5.2 Hashes

**Hash \*NewHash()**

Creates a new empty hash table.

**Hash \*Copy(Hash \*h)**

Make a shallow copy of the hash h.

**void Delete(Hash \*h)**

Deletes h.

**int Len(Hash \*h)**

Returns the number of items in h.

**Object \*Getattr(Hash \*h, String\_or\_char \*key)**

Gets an object from h. key may be a string or a simple char \* string. Returns NULL if not found.

**int Setattr(Hash \*h, String\_or\_char \*key, Object\_or\_char \*val)**

Stores val in h. key may be a string or a simple char \*. If val is not a standard object (String, Hash, or List) it is assumed to be a char \* in which case it is used to construct a String that is stored in the hash. If val is NULL, the object is deleted. Increases the reference count of val. Returns 1 if this operation replaced an existing hash entry, 0 otherwise.

**int Delattr(Hash \*h, String\_or\_char \*key)**

Deletes the hash item referenced by key. Decreases the reference count on the corresponding object (if any). Returns 1 if an object was removed, 0 otherwise.

**List \*Keys(Hash \*h)**

Returns the list of hash table keys.

## 29.5.3 Lists

**List \*NewList()**

Creates a new empty list.

**List \*Copy(List \*x)**

Make a shallow copy of the List x.

**void Delete(List \*x)**

Deletes `x`.

```
int Len(List *x)
```

Returns the number of items in `x`.

```
Object *Getitem(List *x, int n)
```

Returns an object from `x` with index `n`. If `n` is beyond the end of the list, the last item is returned. If `n` is negative, the first item is returned.

```
int *Setitem(List *x, int n, Object_or_char *val)
```

Stores `val` in `x`. If `val` is not a standard object (String, Hash, or List) it is assumed to be a `char *` in which case it is used to construct a String that is stored in the list. `n` must be in range. Otherwise, an assertion will be raised.

```
int *Delitem(List *x, int n)
```

Deletes item `n` from the list, shifting items down if necessary. To delete the last item in the list, use the special value `DOH_END` for `n`.

```
void Append(List *x, Object_or_char *t)
```

Appends `t` to the end of `x`. If `t` is not a standard object, it is assumed to be a `char *` and is used to create a String object.

```
void Insert(String *s, int pos, Object_or_char *t)
```

Inserts `t` into `s` at position `pos`. The contents of `s` are shifted accordingly. The special value `DOH_END` can be used for `pos` to indicate insertion at the end of the list (appending). If `t` is not a standard object, it is assumed to be a `char *` and is used to create a String object.

## 29.5.4 Common operations

The following operations are applicable to all datatypes.

```
Object *Copy(Object *x)
```

Make a copy of the object `x`.

```
void Delete(Object *x)
```

Deletes `x`.

```
void Setfile(Object *x, String_or_char *f)
```

Sets the filename associated with `x`. Used to track objects and report errors.

```
String *Getfile(Object *x)
```

Gets the filename associated with `x`.

```
void Setline(Object *x, int n)
```

Sets the line number associated with `x`. Used to track objects and report errors.

```
int Getline(Object *x)
```

Gets the line number associated with x.

## 29.5.5 Iterating over Lists and Hashes

To iterate over the elements of a list or a hash table, the following functions are used:

```
Iterator First(Object *x)
```

Returns an iterator object that points to the first item in a list or hash table. The `item` attribute of the `Iterator` object is a pointer to the item. For hash tables, the `key` attribute of the `Iterator` object additionally points to the corresponding Hash table key. The `item` and `key` attributes are `NULL` if the object contains no items or if there are no more items.

```
Iterator Next(Iterator i)
```

Returns an iterator that points to the next item in a list or hash table.

Here are two examples of iteration:

```
List *l = (some list);
Iterator i;

for (i = First(l); i.item; i = Next(i)) {
    Printf(stdout, "%s\n", i.item);
}

Hash *h = (some hash);
Iterator j;

for (j = First(j); j.item; j= Next(j)) {
    Printf(stdout, "%s : %s\n", j.key, j.item);
}
```

## 29.5.6 I/O

Special I/O functions are used for all internal I/O. These operations work on `C FILE *` objects, `String` objects, and special `File` objects (which are merely a wrapper around `FILE *`).

```
int Printf(String_or_FILE *f, const char *fmt, ...)
```

Formatted I/O. Same as the C `fprintf()` function except that output can also be directed to a string object. Note: the `%s` format specifier works with both strings and `char *`. All other format operators have the same meaning.

```
int Printv(String_or_FILE *f, String_or_char *arg1,..., NULL)
```

Prints a variable number of strings arguments to the output. The last argument to this function must be `NULL`. The other arguments can either be `char *` or string objects.

```
int Putc(int ch, String_or_FILE *f)
```

Same as the C `fputc()` function.

```
int Write(String_or_FILE *f, void *buf, int len)
```

Same as the C `write()` function.

```
int Read(String_or_FILE *f, void *buf, int maxlen)
```

Same as the C `read()` function.

```
int Getc(String_or_FILE *f)
```

Same as the C `fgetc()` function.

```
int Ungetc(int ch, String_or_FILE *f)
```

Same as the C `ungetc()` function.

```
int Seek(String_or_FILE *f, int offset, int whence)
```

Same as the C `seek()` function. `offset` is the number of bytes. `whence` is one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.

```
long Tell(String_or_FILE *f)
```

Same as the C `tell()` function.

```
File *NewFile(const char *filename, const char *mode)
```

Create a File object using the `fopen()` library call. This file differs from `FILE *` in that it can be placed in the standard SWIG containers (lists, hashes, etc.).

```
File *NewFileFromFile(FILE *f)
```

Create a File object wrapper around an existing `FILE *` object.

```
int Close(String_or_FILE *f)
```

Closes a file. Has no effect on strings.

The use of the above I/O functions and strings play a critical role in SWIG. It is common to see small code fragments of code generated using code like this:

```
/* Print into a string */
String *s = NewString("");
Printf(s, "Hello\n");
for (i = 0; i < 10; i++) {
    Printf(s, "%d\n", i);
}
...
/* Print string into a file */
Printf(f, "%s\n", s);
```

Similarly, the preprocessor and parser all operate on string-files.

## 29.6 Navigating and manipulating parse trees

Parse trees are built as collections of hash tables. Each node is a hash table in which arbitrary attributes can be stored. Certain attributes in the hash table provide links to other parse tree nodes. The following macros can be used to move around the parse tree.

```
String *nodeType(Node *n)
```

Returns the node type tag as a string. The returned string indicates the type of parse tree node.

**Node \*nextSibling(Node \*n)**

Returns the next node in the parse tree. For example, the next C declaration.

**Node \*previousSibling(Node \*n)**

Returns the previous node in the parse tree. For example, the previous C declaration.

**Node \*firstChild(Node \*n)**

Returns the first child node. For example, if n was a C++ class node, this would return the node for the first class member.

**Node \*lastChild(Node \*n)**

Returns the last child node. You might use this if you wanted to append a new node to the of a class.

**Node \*parentNode(Node \*n)**

Returns the parent of node n. Use this to move up the pass tree.

The following macros can be used to change all of the above attributes. Normally, these functions are only used by the parser. Changing them without knowing what you are doing is likely to be dangerous.

**void set\_nodeType(Node \*n, const String\_or\_char)**

Change the node type. tree node.

**void set\_nextSibling(Node \*n, Node \*s)**

Set the next sibling.

**void set\_previousSibling(Node \*n, Node \*s)**

Set the previous sibling.

**void set\_firstChild(Node \*n, Node \*c)**

Set the first child node.

**void set\_lastChild(Node \*n, Node \*c)**

Set the last child node.

**void set\_parentNode(Node \*n, Node \*p)**

Set the parent node.

The following utility functions are used to alter the parse tree (at your own risk)

**void appendChild(Node \*parent, Node \*child)**

Append a child to parent. The appended node becomes the last child.

```
void deleteNode(Node *node)
```

Deletes a node from the parse tree. Deletion reconnects siblings and properly updates the parent so that sibling nodes are unaffected.

## 29.7 Working with attributes

Since parse tree nodes are just hash tables, attributes are accessed using the `Getattr()`, `Setattr()`, and `Delattr()` operations. For example:

```
int functionHandler(Node *n) {
    String *name      = Getattr(n, "name");
    String *symname    = Getattr(n, "sym:name");
    SwigType *type     = Getattr(n, "type");
    ...
}
```

New attributes can be freely attached to a node as needed. However, when new attributes are attached during code generation, they should be prepended with a namespace prefix. For example:

```
...
Setattr(n, "python:docstring", doc);    /* Store docstring */
...
```

A quick way to check the value of an attribute is to use the `checkAttribute()` function like this:

```
if (checkAttribute(n, "storage", "virtual")) {
    /* n is virtual */
    ...
}
```

Changing the values of existing attributes is allowed and is sometimes done to implement node transformations. However, if a function/method modifies a node, it is required to restore modified attributes to their original values. To simplify the task of saving/restoring attributes, the following functions are used:

```
int Swig_save(const char *ns, Node *n, const char *name1, const char *name2, ..., NIL)
```

Saves a copy of attributes `name1`, `name2`, etc. from node `n`. Copies of the attributes are actually resaved in the node in a different namespace which is set by the `ns` argument. For example, if you call `Swig_save("foo", n, "type", NIL)`, then the "type" attribute will be copied and saved as "foo:type". The namespace name itself is stored in the "view" attribute of the node. If necessary, this can be examined to find out where previous values of attributes might have been saved.

```
int Swig_restore(Node *n)
```

Restores the attributes saved by the previous call to `Swig_save()`. Those attributes that were supplied to `Swig_save()` will be restored to their original values.

The `Swig_save()` and `Swig_restore()` functions must always be used as a pair. That is, every call to `Swig_save()` must have a matching call to `Swig_restore()`. Calls can be nested if necessary. Here is an example that shows how the functions might be used:

```
int variableHandler(Node *n) {
    Swig_save("variableHandler", n, "type", "sym:name", NIL);
    String *symname = Getattr(n, "sym:name");
    SwigType *type   = Getattr(n, "type");
    ...
    Append(symname, "_global");           // Change symbol name
    SwigType_add_pointer(type);           // Add pointer
}
```

```

...
generate wrappers
...
Swig_restore(n);           // Restore original values
return SWIG_OK;
}

```

```

int Swig_require(const char *ns, Node *n, const char *name1, const char *name2, ...,
NIL)

```

This is an enhanced version of `Swig_save()` that adds error checking. If an attribute name is not present in `n`, a failed assertion results and SWIG terminates with a fatal error. Optionally, if an attribute name is specified as `"*name"`, a copy of the attribute is saved as with `Swig_save()`. If an attribute is specified as `"?name"`, the attribute is optional. `Swig_restore()` must always be called after using this function.

## 29.8 Type system

SWIG implements the complete C++ type system including typedef, inheritance, pointers, references, and pointers to members. A detailed discussion of type theory is impossible here. However, let's cover the highlights.

### 29.8.1 String encoding of types

All types in SWIG consist of a base datatype and a collection of type operators that are applied to the base. A base datatype is almost always some kind of primitive type such as `int` or `double`. The operators consist of things like pointers, references, arrays, and so forth. Internally, types are represented as strings that are constructed in a very precise manner. Here are some examples:

C datatype	SWIG encoding (strings)
<code>int</code>	<code>"int"</code>
<code>int *</code>	<code>"p.int"</code>
<code>const int *</code>	<code>"p.q(const).int"</code>
<code>int (*x)(int,double)</code>	<code>"p.f(int,double).int"</code>
<code>int [20][30]</code>	<code>"a(20).a(30).int"</code>
<code>int (F::*)(int)</code>	<code>"m(F).f(int).int"</code>
<code>vector&lt;int&gt; *</code>	<code>"p.vector&lt;(int)&gt;"</code>

Reading the SWIG encoding is often easier than figuring out the C code---just read it from left to right. For a type of `"p.f(int,double).int"` is a "pointer to a function(int,double) that returns int".

The following operator encodings are used in type strings:

Operator	Meaning
<code>p.</code>	Pointer to
<code>a(n).</code>	Array of dimension n
<code>r.</code>	C++ reference
<code>m(class).</code>	Member pointer to class
<code>f(args).</code>	Function.
<code>q(qlist).</code>	Qualifiers

In addition, type names may be parameterized by templates. This is represented by enclosing the template parameters in `< ( ... ) >`. Variable length arguments are represented by the special base type of `v ( ... )`.

If you want to experiment with type encodings, the raw type strings can be inserted into an interface file using backticks `` wherever a type is expected. For instance, here is an extremely perverted example:

```
`p.a(10).p.f(int,p.f(int).int)` foo(int, int (*x)(int));
```

This corresponds to the immediately obvious C declaration:

```
(*(*foo(int,int (*) (int)))[10])(int,int (*) (int));
```

Aside from the potential use of this declaration on a C programming quiz, it motivates the use of the special SWIG encoding of types. The SWIG encoding is much easier to work with because types can be easily examined, modified, and constructed using simple string operations (comparison, substrings, concatenation, etc.). For example, in the parser, a declaration like this

```
int a[30];
```

is processed in a few pieces. In this case, you have the base type "int" and the declarator of type "a ( 30 ) . p . ". To make the final type, the two parts are just joined together using string concatenation.

## 29.8.2 Type construction

The following functions are used to construct types. You should use these functions instead of trying to build the type strings yourself.

```
void SwigType_add_pointer(SwigType *ty)
```

Adds a pointer to `ty`.

```
void SwigType_del_pointer(SwigType *ty)
```

Removes a single pointer from `ty`.

```
void SwigType_add_reference(SwigType *ty)
```

Adds a reference to `ty`.

```
void SwigType_add_array(SwigType *ty, String_or_char *dim)
```

Adds an array with dimension `dim` to `ty`.

```
void SwigType_add_qualifier(SwigType *ty, String_or_char *q)
```

Adds a type qualifier `q` to `ty`. `q` is typically "const" or "volatile".

```
void SwigType_add_memberpointer(SwigType *ty, String_or_char *cls)
```

Adds a pointer to a member of class `cls` to `ty`.

```
void SwigType_add_function(SwigType *ty, ParmList *p)
```

Adds a function to `ty`. `p` is a linked-list of parameter nodes as generated by the parser. See the section on parameter lists for details about the representation.

```
void SwigType_add_template(SwigType *ty, ParmList *p)
```

Adds a template to `ty`. `p` is a linked-list of parameter nodes as generated by the parser. See the section on parameter lists for details about the representation.

```
SwigType *SwigType_pop(SwigType *ty)
```

Removes the last type constructor from `ty` and returns it. `ty` is modified.

```
void SwigType_push(SwigType *ty, SwigType *op)
```



Pushes the type operators in `op` onto type `ty`. The opposite of `SwigType_pop()`.

**SwigType \*SwigType\_pop\_arrays(SwigType \*ty)**

Removes all leading array operators from `ty` and returns them. `ty` is modified. For example, if `ty` is `"a(20).a(10).p.int"`, then this function would return `"a(20).a(10)."` and modify `ty` so that it has the value `"p.int"`.

**SwigType \*SwigType\_pop\_function(SwigType \*ty)**

Removes a function operator from `ty` including any qualification. `ty` is modified. For example, if `ty` is `"f(int).int"`, then this function would return `"f(int)."` and modify `ty` so that it has the value `"int"`.

**SwigType \*SwigType\_base(SwigType \*ty)**

Returns the base type of a type. For example, if `ty` is `"p.a(20).int"`, this function would return `"int"`. `ty` is unmodified.

**SwigType \*SwigType\_prefix(SwigType \*ty)**

Returns the prefix of a type. For example, if `ty` is `"p.a(20).int"`, this function would return `"p.a(20)."`. `ty` is unmodified.

### 29.8.3 Type tests

The following functions can be used to test properties of a datatype.

**int SwigType\_ispointer(SwigType \*ty)**

Checks if `ty` is a standard pointer.

**int SwigType\_ismemberpointer(SwigType \*ty)**

Checks if `ty` is a member pointer.

**int SwigType\_isreference(SwigType \*ty)**

Checks if `ty` is a C++ reference.

**int SwigType\_isarray(SwigType \*ty)**

Checks if `ty` is an array.

**int SwigType\_isfunction(SwigType \*ty)**

Checks if `ty` is a function.

**int SwigType\_isqualifier(SwigType \*ty)**

Checks if `ty` is a qualifier.

**int SwigType\_issimple(SwigType \*ty)**

Checks if `ty` is a simple type. No operators applied.

**int SwigType\_isconst(SwigType \*ty)**

Checks if `ty` is a const type.

```
int SwigType_isvarargs(SwigType *ty)
```

Checks if `ty` is a varargs type.

```
int SwigType_istemplate(SwigType *ty)
```

Checks if `ty` is a templated type.

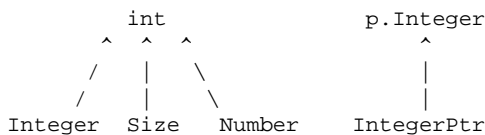
## 29.8.4 Typedef and inheritance

The behavior of typedef declaration is to introduce a type alias. For instance, `typedef int Integer` makes the identifier `Integer` an alias for `int`. The treatment of typedef in SWIG is somewhat complicated due to the pattern matching rules that get applied in typemaps and the fact that SWIG prefers to generate wrapper code that closely matches the input to simplify debugging (a user will see the typedef names used in their program instead of the low-level primitive C datatypes).

To handle typedef, SWIG builds a collection of trees containing typedef relations. For example,

```
typedef int Integer;
typedef Integer *IntegerPtr;
typedef int Number;
typedef int Size;
```

produces two trees like this:



To resolve a single typedef relationship, the following function is used:

```
SwigType *SwigType_typedef_resolve(SwigType *ty)
```

Checks if `ty` can be reduced to a new type via typedef. If so, returns the new type. If not, returns NULL.

Typedefs are only resolved in simple typenames that appear in a type. For example, the type base name and in function parameters. When resolving types, the process starts in the leaf nodes and moves up the tree towards the root. Here are a few examples that show how it works:

Original type	After typedef_resolve()
Integer	int
a(30).Integer	int
p.IntegerPtr	p.p.Integer
p.p.Integer	p.p.int

For complicated types, the process can be quite involved. Here is the reduction of a function pointer:

```

p.f(Integer, p.IntegerPtr, Size).Integer      : Start
p.f(Integer, p.IntegerPtr, Size).int
p.f(int, p.IntegerPtr, Size).int
p.f(int, p.p.Integer, Size).int
p.f(int, p.p.int, Size).int
p.f(int, p.p.int, int).int                    : End
  
```

Two types are equivalent if their full type reductions are the same. The following function will fully reduce a datatype:

```
SwigType *SwigType_typedef_resolve_all(SwigType *ty)
```

Fully reduces `ty` according to typedef rules. Resulting datatype will consist only of primitive typenames.

## 29.8.5 Lvalues

When generating wrapper code, it is necessary to emit datatypes that can be used on the left-hand side of an assignment operator (an lvalue). However, not all C datatypes can be used in this way—especially arrays and const-qualified types. To generate a type that can be used as an lvalue, use the following function:

```
SwigType *SwigType_ltype(SwigType *ty)
```

Converts type `ty` to a type that can be used as an lvalue in assignment. The resulting type is stripped of qualifiers and arrays are converted to a pointers.

The creation of lvalues is fully aware of typedef and other aspects of the type system. Therefore, the creation of an lvalue may result in unexpected results. Here are a few examples:

```
typedef double Matrix4[4][4];
Matrix4 x;    // type = 'Matrix4', ltype='p.a(4).double'

typedef const char * Literal;
Literal y;    // type = 'Literal', ltype='p.char'
```

## 29.8.6 Output functions

The following functions produce strings that are suitable for output.

```
String *SwigType_str(SwigType *ty, String_or_char *id = 0)
```

Generates a C string for a datatype. `id` is an optional declarator. For example, if `ty` is `"p.f(int).int"` and `id` is `"foo"`, then this function produces `"int (*foo)(int)"`. This function is used to convert string-encoded types back into a form that is valid C syntax.

```
String *SwigType_lstr(SwigType *ty, String_or_char *id = 0)
```

This is the same as `SwigType_str()` except that the result is generated from the type's lvalue (as generated from `SwigType_ltype`).

```
String *SwigType_lcaststr(SwigType *ty, String_or_char *id = 0)
```

Generates a casting operation that converts from type `ty` to its lvalue. `id` is an optional name to include in the cast. For example, if `ty` is `"q(const).p.char"` and `id` is `"foo"`, this function produces the string `"(char *) foo"`.

```
String *SwigType_rcaststr(SwigType *ty, String_or_char *id = 0)
```

Generates a casting operation that converts from a type's lvalue to a type equivalent to `ty`. `id` is an optional name to include in the cast. For example, if `ty` is `"q(const).p.char"` and `id` is `"foo"`, this function produces the string `"(const char *) foo"`.

```
String *SwigType_manglestr(SwigType *ty)
```

Generates a mangled string encoding of type `ty`. The mangled string only contains characters that are part of a valid C identifier. The resulting string is used in various parts of SWIG, but is most commonly associated with type-descriptor objects that appear in wrappers (e.g., `SWIGTYPE_p_double`).

## 29.9 Parameters

Several type-related functions involve parameter lists. These include functions and templates. Parameter list are represented as a list of nodes with the following attributes:

"type"	-	Parameter type	(required)
"name"	-	Parameter name	(optional)
"value"	-	Initializer	(optional)

Typically parameters are denoted in the source by using a typename of `Parm *` or `ParmList *`. To walk a parameter list, simply use code like this:

```
Parm *parms;
Parm *p;
for (p = parms; p; p = nextSibling(p)) {
    SwigType *type = Getattr(p, "type");
    String *name = Getattr(p, "name");
    String *value = Getattr(p, "value");
    ...
}
```

Note: this code is exactly the same as what you would use to walk parse tree nodes.

An empty list of parameters is denoted by a NULL pointer.

Since parameter lists are fairly common, the following utility functions are provided to manipulate them:

**Parm \*CopyParm(Parm \*p);**

Copies a single parameter.

**ParmList \*CopyParmList(ParmList \*p);**

Copies an entire list of parameters.

**int ParmList\_len(ParmList \*p);**

Returns the number of parameters in a parameter list.

**String \*ParmList\_str(ParmList \*p);**

Converts a parameter list into a C string. For example, produces a string like `"(int *p, int n, double x);"`.

**String \*ParmList\_protostr(ParmList \*p);**

The same as `ParmList_str()` except that parameter names are not included. Used to emit prototypes.

**int ParmList\_numrequired(ParmList \*p);**

Returns the number of required (non-optional) arguments in `p`.

## 29.10 Writing a Language Module

This section briefly outlines the steps needed to create a bare-bones language module. For more advanced techniques, you should look at the implementation of existing modules. Since the code is relatively easy to read, this section describes the creation of a minimal Python module. You should be able to extrapolate this to other languages.

## 29.10.1 Execution model

Code generation modules are defined by inheriting from the `Language` class, currently defined in the `Source/Modules1.1` directory of SWIG. Starting from the parsing of command line options, all aspects of code generation are controlled by different methods of the `Language` that must be defined by your module.

## 29.10.2 Starting out

To define a new language module, first create a minimal implementation using this example as a guide:

```
#include "swigmod.h"

#ifdef MACSWIG
#include "swigconfig.h"
#endif

class PYTHON : public Language {
public:

    virtual void main(int argc, char *argv[]) {
        printf("I'm the Python module.\n");
    }

    virtual int top(Node *n) {
        printf("Generating code.\n");
        return SWIG_OK;
    }

};

extern "C" Language *
swig_python(void) {
    return new PYTHON();
}
```

The "swigmod.h" header file contains, among other things, the declaration of the `Language` base class and so you should include it at the top of your language module's source file. Similarly, the "swigconfig.h" header file contains some other useful definitions that you may need. Note that you should *not* include any header files that are installed with the target language. That is to say, the implementation of the SWIG Python module shouldn't have any dependencies on the Python header files. The wrapper code generated by SWIG will almost always depend on some language-specific C/C++ header files, but SWIG itself does not.

Give your language class a reasonable name, usually the same as the target language. By convention, these class names are all uppercase (e.g. "PYTHON" for the Python language module) but this is not a requirement. This class will ultimately consist of a number of overrides of the virtual functions declared in the `Language` base class, in addition to any language-specific member functions and data you need. For now, just use the dummy implementations shown above.

The language module ends with a factory function, `swig_python()`, that simply returns a new instance of the language class. As shown, it should be declared with the `extern "C"` storage qualifier so that it can be called from C code. It should also return a pointer to the base class (`Language`) so that only the interface (and not the implementation) of your language module is exposed to the rest of SWIG.

Save the code for your language module in a file named "python.cxx" and place this file in the `Source/Modules1.1` directory of the SWIG distribution. To ensure that your module is compiled into SWIG along with the other language modules, modify the file `Source/Modules1.1/Makefile.in` to include the additional source files. Look for the lines that define the `SRCS` and `OBJS` variables and add entries for your language. In addition, modify the file `Source/Modules1.1/swigmain.cxx` with an additional command line option that activates the module. Read the source—it's straightforward.

Next, at the top level of the SWIG distribution, re-run the `autogen.sh` script to regenerate the various build files:

```
$ sh autogen.sh
```

Next re-run `configure` to regenerate all of the Makefiles:

```
$ ./configure
```

Finally, rebuild SWIG with your module added:

```
$ make
```

Once it finishes compiling, try running SWIG with the command-line option that activates your module. For example, `swig -python foo.i`. The messages from your new module should appear.

### 29.10.3 Command line options

When SWIG starts, the command line options are passed to your language module. This occurs before any other processing occurs (preprocessing, parsing, etc.). To capture the command line options, simply use code similar to this:

```
void Language::main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        if (argv[i]) {
            if (strcmp(argv[i], "-interface") == 0) {
                if (argv[i+1]) {
                    interface = NewString(argv[i+1]);
                    Swig_mark_arg(i);
                    Swig_mark_arg(i+1);
                    i++;
                } else {
                    Swig_arg_error();
                }
            } else if (strcmp(argv[i], "-globals") == 0) {
                if (argv[i+1]) {
                    global_name = NewString(argv[i+1]);
                    Swig_mark_arg(i);
                    Swig_mark_arg(i+1);
                    i++;
                } else {
                    Swig_arg_error();
                }
            } else if ( (strcmp(argv[i], "-proxy") == 0) ) {
                proxy_flag = 1;
                Swig_mark_arg(i);
            } else if (strcmp(argv[i], "-keyword") == 0) {
                use_kw = 1;
                Swig_mark_arg(i);
            } else if (strcmp(argv[i], "-help") == 0) {
                fputs(usage, stderr);
            }
            ...
        }
    }
}
```

The exact set of options depends on what you want to do in your module. Generally, you would use the options to change code generation modes or to print diagnostic information.

If a module recognizes an option, it should always call `Swig_mark_arg()` to mark the option as valid. If you forget to do this, SWIG will terminate with an unrecognized command line option error.

## 29.10.4 Configuration and preprocessing

In addition to looking at command line options, the `main()` method is responsible for some initial configuration of the SWIG library and preprocessor. To do this, insert some code like this:

```
void main(int argc, char *argv[]) {
    ... command line options ...

    /* Set language-specific subdirectory in SWIG library */
    SWIG_library_directory("python");

    /* Set language-specific preprocessing symbol */
    Preprocessor_define("SWIGPYTHON 1", 0);

    /* Set language-specific configuration file */
    SWIG_config_file("python.swg");

    /* Set typemap language (historical) */
    SWIG_typemap_lang("python");
}
```

The above code does several things—it registers the name of the language module with the core, it supplies some preprocessor macro definitions for use in input files (so that they can determine the target language), and it registers a start-up file. In this case, the file `python.swg` will be parsed before any part of the user-supplied input file.

Before proceeding any further, create a directory for your module in the SWIG library (The `Lib` directory). Now, create a configuration file in the directory. For example, `python.swg`.

Just to review, your language module should now consist of two files—an implementation file `python.cxx` and a configuration file `python.swg`.

## 29.10.5 Entry point to code generation

SWIG is a multi-pass compiler. Once the `main()` method has been invoked, the language module does not execute again until preprocessing, parsing, and a variety of semantic analysis passes have been performed. When the core is ready to start generating wrappers, it invokes the `top()` method of your language class. The argument to `top` is a single parse tree node that corresponds to the top of the entire parse tree.

To get the code generation process started, the `top()` procedure needs to do several things:

- Initialize the wrapper code output.
- Set the module name.
- Emit common initialization code.
- Emit code for all of the child nodes.
- Finalize the wrapper module and cleanup.

An outline of `top()` might be as follows:

```
int Python::top(Node *n) {

    /* Get the module name */
    String *module = Getattr(n,"name");

    /* Get the output file name */
    String *outfile = Getattr(n,"outfile");

    /* Initialize I/O (see next section) */
    ...

    /* Output module initialization code */
    ...
}
```

```

/* Emit code for children */
Language::top(n);

...
/* Cleanup files */
...

return SWIG_OK;
}

```

## 29.10.6 Module I/O and wrapper skeleton

## 29.10.7 Low-level code generators

## 29.10.8 Configuration files

At the time of this writing, SWIG supports nearly a dozen languages, which means that for continued sanity in maintaining the configuration files, the language modules need to follow some conventions. These are outlined here along with the admission that, yes it is ok to violate these conventions in minor ways, as long as you know where to apply the proper kludge to keep the overall system regular and running. Engineering is the art of compromise, see...

Much of the maintenance regularity depends on choosing a suitable nickname for your language module (and then using it in a controlled way). Nicknames should be all lower case letters with an optional numeric suffix (no underscores, no dashes, no spaces). Some examples are: `foo`, `bar`, `qux99`.

The numeric suffix variant, as in the last example, is somewhat tricky to work with because sometimes people expect to refer to the language without this number but sometimes that number is extremely relevant (especially when it corresponds to language implementation versions with incompatible interfaces). New language modules that unavoidably require a numeric suffix in their nickname should include that number in all uses, or be prepared to kludge.

The nickname is used in four places:

<b>usage</b>	<b>transform</b>
"skip" tag	(none)
Examples/ subdir name	(none)
Examples/GIFPlot/ subdir name	capitalize (upcase first letter)
Examples/test-suite/ subdir name	(none)

As you can see, most usages are direct.

### *configure.in*

This file is processed by [autoconf](#) to generate the `configure` script. This is where you need to add shell script fragments and `autoconf` macros to detect the presence of whatever development support your language module requires, typically directories where headers and libraries can be found, and/or utility programs useful for integrating the generated wrapper code.

Use the `AC_ARG_WITH`, `AC_MSG_CHECKING`, `AC_SUBST` macros and so forth (see other languages for examples). Avoid using the `[` and `]` character in shell script fragments. The variable names passed to `AC_SUBST` should begin with the nickname, entirely upcased.

At the end of the new section is the place to put the aforementioned nickname kludges (should they be needed). See Perl5 and Php4 for examples of what to do. [If this is still unclear after you've read the code, ping me and I'll expand on this further. ---ttn]

### *Makefile.in*



Some of the variables `AC_SUBSTituted` are essential to the support of your language module. Fashion these into a shell script "test" clause and assign that to a skip tag using "-z" and "-o":

```
skip-qux99 = [ -z "@QUX99INCLUDE@" -o -z "@QUX99LIBS" ]
```

This means if those vars should ever be empty, qux99 support should be considered absent and so it would be a good idea to skip actions that might rely on it.

Here is where you may also define an alias (but then you'll need to kludge --- don't do this):

```
skip-qux = $(skip-qux99)
```

Lastly, you need to modify each of `check-aliveness`, `check-examples`, `check-test-suite`, `check-gifplot` (all targets) and `lib-languages` (var). Use the nickname for these, not the alias. Note that you can do this even before you have any tests or examples set up; the Makefile rules do some sanity checking and skip around these kinds of problems.

### *Examples/Makefile.in*

Nothing special here; see comments at top the of this file and look to the existing languages for examples.

### *Examples/qux99/check.list*

Do `cp ../python/check.list .` and modify to taste. One subdir per line.

### *Examples/GLIFPlot/Qux99/check.list*

Do `cp ../Python/check.list .` and modify to taste. One subdir per line.

### *Lib/qux99/extra-install.list*

If you add your language to the top-level `Makefile.in` var `lib-languages`, then `make install` will install all `*.i` and `*.swg` files from the language-specific subdirectory of `Lib`. Use (optional) file `extra-install.list` in that directory to name additional files to install (see `ruby` for example).

### *Runtime/Makefile.in*

Add another `make` invocation to `all`, and a section for your language module.

### *Source/Modules1.1/Makefile.in*

Add appropriate entries for vars `OBJS` and `SRCS`. That's it!

At some point it would be a good idea to use [automake](#) to handle some of these configuration tasks, but that point is now long past. If you are interested in working on that, feel free to raise the issue in the context of a next-generation clean-slate SWIG.

## 29.10.9 Runtime support

Discuss the kinds of functions typically needed for SWIG runtime support (e.g. `SWIG_ConvertPtr()` and `SWIG_NewPointerObj()`) and the names of the SWIG files that implement those functions.

### 29.10.10 Standard library files

Discuss the standard library files that most language modules provide, e.g.

- `typemaps.i`
- `std_string.i`
- `std_vector.i`
- `stl.i`

### 29.10.11 Examples and test cases

Each of the language modules provides one or more examples. These examples are used to demonstrate different features of the language module to SWIG end-users, but you'll find that they're useful during development and testing of your language module as well. You can use examples from the existing SWIG language modules for inspiration.

Each example is self-contained and consists of (at least) a `Makefile`, a SWIG interface file for the example module, and a script that demonstrates the functionality for that module. All of these files are stored in the same subdirectory, and that directory

should be nested under `Examples/python`. For example, the files for the Python "simple" example are found in `Examples/python/simple`.

By default, all of the examples are built and run when the user types `make check`. To ensure that your examples are automatically run during this process, see the section on [configuration files](#).

## 29.10.12 Documentation

Don't forget to write end-user documentation for your language module. Currently, each language module has a dedicated chapter (although this structure may change in the future). You shouldn't rehash things that are already covered in sufficient detail in the [SWIG Basics](#) and [SWIG and C++](#) chapters. There is no fixed format for *what*, exactly, you should document about your language module, but you'll obviously want to cover issues that are unique to your language.

Some topics that you'll want to be sure to address include:

- Command line options unique to your language module.
- Non-obvious mappings between C/C++ and scripting language concepts. For example, if your scripting language provides a single floating point type, it should be no big surprise to find that C/C++ `float` and `double` types are mapped to it. On the other hand, if your scripting language doesn't provide support for "classes" or something similar, you'd want to discuss how C++ classes are handled.
- How to compile the SWIG-generated wrapper code into shared libraries that can actually be used. For some languages, there are well-defined procedures for doing this, but for others it's an ad hoc process. Provide as much detail as appropriate, and links to other resources if available.

## 29.11 Typemaps

### 29.11.1 Proxy classes

## 29.12 Guide to parse tree nodes

This section describes the different parse tree nodes and their attributes.

### **cdecl**

Describes general C declarations including variables, functions, and typedefs. A declaration is parsed as "storage T D" where storage is a storage class, T is a base type, and D is a declarator.

"name"	- Declarator name
"type"	- Base type T
"decl"	- Declarator type (abstract)
"storage"	- Storage class (static, extern, typedef, etc.)
"parms"	- Function parameters (if a function)
"code"	- Function body code (if supplied)
"value"	- Default value (if supplied)

### **constructor**

C++ constructor declaration.

"name"	- Name of constructor
"parms"	- Parameters
"decl"	- Declarator (function with parameters)
"code"	- Function body code (if any)
"feature:new"	- Set to indicate return of new object.

### **destructor**

C++ destructor declaration.

```
"name"          - Name of destructor
"code"          - Function body code (if any)
"storage"       - Storage class (set if virtual)
"value"        - Default value (set if pure virtual).
```

## access

C++ access change.

```
"kind"          - public, protected, private
```

## constant

Constant created by %constant or #define.

```
"name"          - Name of constant.
"type"          - Base type.
"value"         - Value.
"storage"       - Set to %constant
"feature:immutable" - Set to indicate read-only
```

## class

C++ class definition or C structure definition.

```
"name"          - Name of the class.
"kind"          - Class kind ("struct", "union", "class")
"symtab"       - Enclosing symbol table.
"tdname"       - Typedef name. Use for typedef struct { ... } A.
"abstract"     - Set if class has pure virtual methods.
"baselist"     - List of base class names.
"storage"      - Storage class (if any)
"unnamed"     - Set if class is unnamed.
```

## enum

Enumeration.

```
"name"          - Name of the enum (if supplied).
"storage"       - Storage class (if any)
"tdname"       - Typedef name (typedef enum { ... } name).
"unnamed"     - Set if enum is unnamed.
```

## enumitem

Enumeration value.

```
"name"          - Name of the enum value.
"type"          - Type (integer or char)
"value"         - Enum value (if given)
"feature:immutable" - Set to indicate read-only
```

## namespace

C++ namespace.

```
"name"          - Name of the namespace.
"symtab"       - Symbol table for enclosed scope.
"unnamed"     - Set if unnamed namespace
```

"alias" - Alias name. Set for namespace A = B;

## using

C++ using directive.

"name" - Name of the object being referred to.  
 "uname" - Qualified name actually given to using.  
 "node" - Node being referenced.  
 "namespace" - Namespace name being reference (using namespace name)

## classforward

A forward C++ class declaration.

"name" - Name of the class.  
 "kind" - Class kind ("union", "struct", "class")

## insert

Code insertion directive. For example, %{ ... %} or %insert(section).

"code" - Inserted code  
 "section" - Section name ("header", "wrapper", etc.)

## top

Top of the parse tree.

"module" - Module name

## extend

%extend directive.

"name" - Module name  
 "symtab" - Symbol table of enclosed scope.

## apply

%apply pattern { patternlist }.

"pattern" - Source pattern.  
 "symtab" - Symbol table of enclosed scope.

## clear

%clear patternlist;

"firstChild" - Patterns to clear

## include

%include directive.

"name" - Filename  
 "firstChild" - Children

## import

**%import directive.**

```
"name"      - Filename
"firstChild" - Children
```

## module

**%module directive.**

```
"name"      - Name of the module
```

## typemap

**%typemap directive.**

```
"method"      - Typemap method name.
"code"        - Typemap code.
"kwargs"      - Keyword arguments (if any)
"firstChild"  - Typemap patterns
```

## typemapcopy

**%typemap directive with copy.**

```
"method"      - Typemap method name.
"pattern"     - Typemap source pattern.
"firstChild"  - Typemap patterns
```

## typemapitem

**%typemap pattern.** Used with %apply, %clear, %typemap.

```
"pattern"     - Typemap pattern (a parameter list)
"parms"       - Typemap parameters.
```

## types

**%types directive.**

```
"parms"       - List of parameter types.
```

## extern

**extern "X" { ... } declaration.**

```
"name"       - Name "C", "Fortran", etc.
```