

SWIG1.3 Documentation

Table of Contents

<u>SWIG1.3 Development Documentation</u>	1
<u>Documentation updated to SWIG-1.3</u>	1
<u>Documentation partially updated to SWIG-1.3</u>	1
<u>Documentation that has not yet been updated</u>	1
<u>Documentation not yet written</u>	2
<u>Preface</u>	3
<u>Introduction</u>	3
<u>Special Introduction for Version 1.3</u>	3
<u>SWIG Versions</u>	3
<u>SWIG resources</u>	3
<u>Prerequisites</u>	4
<u>Organization of this manual</u>	4
<u>How to avoid reading the manual</u>	4
<u>Credits</u>	4
<u>Bug reports</u>	4
<u>1 Introduction</u>	7
<u>What is SWIG?</u>	7
<u>Why use SWIG?</u>	7
<u>A SWIG example</u>	8
<u>SWIG interface file</u>	8
<u>The swig command</u>	9
<u>Building a Perl5 module</u>	9
<u>Building a Python module</u>	9
<u>Shortcuts</u>	10
<u>Building libraries and modules</u>	10
<u>C syntax, but not a C compiler</u>	11
<u>Non-intrusive interface building</u>	11
<u>Hands off code generation</u>	12
<u>Getting started on Windows</u>	13
<u>Installation on Windows</u>	13
<u>Windows Executable</u>	13
<u>SWIG Windows Examples</u>	13
<u>Instructions for using the Examples with Visual C++</u>	13
<u>Instructions for using the Examples with other compilers</u>	14
<u>SWIG on Cygwin and Mingw</u>	14
<u>Building swig.exe on Windows</u>	15
<u>Running the examples on Windows using Cygwin and Mingw</u>	15
<u>2 Scripting Languages</u>	17
<u>The two language view of the world</u>	17
<u>How does a scripting language talk to C?</u>	17
<u>Wrapper functions</u>	17
<u>Variable linking</u>	18
<u>Constants</u>	19
<u>Structures and classes</u>	19
<u>Shadow classes</u>	20
<u>Building scripting language extensions</u>	20
<u>Shared libraries and dynamic loading</u>	21
<u>Linking with shared libraries</u>	21

Table of Contents

2 Scripting Languages

<u>Static linking</u>	22
-----------------------------	----

3 SWIG Basics.....23

<u>Running SWIG</u>	23
<u>Input format</u>	23
<u>SWIG Output</u>	24
<u>Comments</u>	24
<u>C Preprocessor</u>	24
<u>SWIG Directives</u>	24
<u>Parser Limitations</u>	25
<u>Wrapping Simple C Declarations</u>	26
<u>Basic Type Handling</u>	27
<u>Global Variables</u>	28
<u>Constants</u>	29
<u>A brief word about const</u>	30
<u>Pointers and complex objects</u>	31
<u>Simple pointers</u>	31
<u>Run time pointer type checking</u>	31
<u>Derived types, structs, and classes</u>	32
<u>Undefined datatypes</u>	32
<u>Typedef</u>	33
<u>Other Practicalities</u>	34
<u>Passing structures by value</u>	34
<u>Return by value</u>	34
<u>Linking to structure variables</u>	35
<u>Linking to char *</u>	35
<u>Arrays</u>	36
<u>Creating read-only variables</u>	38
<u>Renaming and ignoring declarations</u>	38
<u>Default/optional arguments</u>	40
<u>Pointers to functions and callbacks</u>	40
<u>Structures and unions</u>	42
<u>Typedef and structures</u>	43
<u>Character strings and structures</u>	43
<u>Array members</u>	44
<u>C constructors and destructors</u>	44
<u>Adding member functions to C structures</u>	45
<u>Nested structures</u>	47
<u>Other things to note about structure wrapping</u>	48
<u>Code Insertion</u>	49
<u>The output of SWIG</u>	49
<u>Code insertion blocks</u>	50
<u>Inlined code blocks</u>	50
<u>Initialization blocks</u>	51
<u>SWIG Preprocessor</u>	51
<u>File inclusion</u>	51
<u>File imports</u>	51
<u>Conditional Compilation</u>	52
<u>Macro Expansion</u>	52
<u>SWIG Macros</u>	53
<u>Preprocessing and % { ... } blocks</u>	53

Table of Contents

3 SWIG Basics

<u>Preprocessing and { ... }</u>	54
<u>An Interface Building Strategy</u>	54
<u>Preparing a C program for SWIG</u>	54
<u>The SWIG interface file</u>	55
<u>Why use separate interface files?</u>	55
<u>Getting the right header files</u>	56
<u>What to do with main()</u>	56
<u>How to avoid creating the interface from hell</u>	56

4 SWIG and C++.....59

<u>Comments on C++ Wrapping</u>	59
<u>Supported C++ features</u>	59
<u>A simple C++ example</u>	60
<u>Constructors and destructors</u>	60
<u>Member functions</u>	61
<u>Static members</u>	62
<u>Member data</u>	62
<u>Protection</u>	62
<u>Enums and constants</u>	62
<u>References and pointers</u>	63
<u>Pass and return by value</u>	63
<u>Inheritance</u>	64
<u>Renaming</u>	66
<u>Wrapping Overloaded Functions and Methods</u>	66
<u>Wrapping overloaded operators</u>	74
<u>Adding new methods</u>	76
<u>Templates</u>	77
<u>Pointers to Members</u>	81
<u>Partial class definitions</u>	81
<u>A brief rant about const-correctness</u>	82
<u>Where to go for more information</u>	83

4 Multiple files and the SWIG library.....85

<u>The %include directive</u>	85
<u>The %import directive</u>	85
<u>Including files on the command line</u>	86
<u>The SWIG library</u>	86
<u>Library example</u>	86
<u>Creating Library Files</u>	87
<u> telsh.i</u>	87
<u> malloc.i</u>	87
<u> Placing the files in the library</u>	88
<u>Working with library files</u>	88
<u> Wrapping a library file</u>	88
<u> Checking out library files</u>	88

5 Documentation System.....91

6 Types and Typemaps.....93

<u>Introduction</u>	93
<u>The Problem</u>	93

Table of Contents

6 Types and Typemaps

<u>Typemaps</u>	94
<u>Managing input and output parameters</u>	96
<u>Input parameters</u>	96
<u>Output parameters</u>	97
<u>Input/Output parameters</u>	97
<u>Using different names</u>	98
<u>Array handling</u>	98
<u>Applying constraints to input values</u>	99
<u>Simple constraint example</u>	99
<u>Constraint methods</u>	99
<u>Applying constraints to new datatypes</u>	99
<u>Writing new typemaps</u>	100
<u>The SWIG type system</u>	100
<u>What is a typemap?</u>	101
<u>Creating a new typemap</u>	102
<u>Deleting a typemap</u>	104
<u>Copying a typemap</u>	104
<u>Typemap matching rules</u>	104
<u>Common typemap methods</u>	105
<u>Writing typemap code</u>	108
<u>Local scope</u>	108
<u>Creating local variables</u>	108
<u>Special variables</u>	110
<u>Typemap Parameters</u>	112
<u>Typemaps for arrays</u>	112
<u>Implementing constraints with typemaps</u>	115
<u>Multi-argument typemaps</u>	116
<u>The default typemaps</u>	119
<u>The run-time type checker</u>	120
<u>More about %apply and %clear</u>	122
<u>Reducing wrapper code size</u>	123
<u>Where to go for more information?</u>	124

7 Exceptions, Features, and other Customizations.....125

<u>Exception handling with %exception</u>	125
<u>Handling exceptions in C code</u>	125
<u>Exception handling with longjmp()</u>	126
<u>Handling C++ exceptions</u>	127
<u>Defining different exception handlers</u>	128
<u>Applying exception handlers to specific datatypes</u>	129
<u>Using The SWIG exception library</u>	130
<u>Object ownership and %newobject</u>	130
<u>Features and the %feature directive</u>	132

8 SWIG and Perl5.....135

<u>Preliminaries</u>	135
<u>Running SWIG</u>	135
<u>Getting the right header files</u>	135
<u>Compiling a dynamic module</u>	135
<u>Building a dynamic module with MakeMaker</u>	136
<u>Building a static version of Perl</u>	136

Table of Contents

8 SWIG and Perl5

Compilation problems and compiling with C++	137
Building Perl Extensions under Windows 95/NT	137
Running SWIG from Developer Studio	138
Using NMAKE	138
Modules, packages, and classes	139
Basic Perl interface	140
Functions	140
Global variables	140
Constants	141
Pointers	141
Structures and C++ classes	142
Examples	142
Exception handling	142
Remapping datatypes with typemaps	144
A simple typemap example	144
Perl5 typemaps	144
Typemap variables	145
Name based type conversion	145
Converting a Perl5 array to a char **	146
Using typemaps to return values	147
Accessing array structure members	148
Turning Perl references into C pointers	149
Useful functions	150
Standard typemaps	150
Pointer handling	150
Return values	151
The gory details on shadow classes	151
Module and package names	151
What gets created?	151
Object Ownership	153
Nested Objects	154
Shadow Functions	155
Inheritance	155
Iterators	156
Where to go from here?	157

9 SWIG and Python.....159

Preliminaries	159
Getting the right header files	159
Compiling a dynamic module	159
Using distutils	160
Static linking	160
Using your module	161
Compilation of C++ extensions	162
Compiling for 64-bit platforms	163
Building Python Extensions under Windows	163
The primitive Python-C interface	164
Modules	164
Functions	165
Variable Linking	165
Constants	166

Table of Contents

9 SWIG and Python

<u>Pointers</u>	166
<u>Structures</u>	168
<u>C++ classes</u>	169
<u>C++ classes and type-checking</u>	170
<u>C++ overloaded functions</u>	171
<u>Operators</u>	171
<u>Input and output parameters</u>	171
<u>Simple exception handling</u>	173
<u>Typemaps</u>	177
<u>What is a typemap?</u>	177
<u>Python typemaps</u>	178
<u>Typemap variables</u>	179
<u>Useful Functions</u>	180
<u>Typemap Examples</u>	181
<u>Converting Python list to a char **</u>	181
<u>Expanding a Python object to multiple arguments</u>	183
<u>Using typemaps to return arguments</u>	183
<u>Mapping Python tuples into small arrays</u>	185
<u>Mapping sequences to C arrays</u>	185
<u>Accessing array structure members</u>	186
<u>Pointer handling</u>	187
<u>Other odds and ends</u>	188
<u>Adding native Python functions to a SWIG module</u>	188
<u>Python shadow classes</u>	189
<u>A simple example</u>	189
<u>Why shadow classes?</u>	189
<u>Automatic shadow class generation</u>	190
<u>Compiling modules with shadow classes</u>	191
<u>Shadow classes and type-checking</u>	191
<u>A preview</u>	192
<u>The gory details of shadow classes</u>	192
<u>A simple shadow class</u>	192
<u>Module names</u>	193
<u>Two classes</u>	193
<u>The this pointer</u>	193
<u>Object ownership</u>	194
<u>Constructors and Destructors</u>	194
<u>Member data</u>	194
<u>Printing</u>	194
<u>Shadow Functions</u>	194
<u>Nested objects</u>	195
<u>Inheritance and shadow classes</u>	197
<u>Methods that return new objects</u>	197
<u>Performance concerns and hints</u>	198

10 SWIG and Tcl.....199

<u>Preliminaries</u>	199
<u>Getting the right header files</u>	199
<u>Compiling a dynamic module</u>	199
<u>Static linking</u>	200
<u>Using your module</u>	200

Table of Contents

10 SWIG and Tcl

Compilation of C++ extensions	202
Compiling for 64-bit platforms	203
Setting a package prefix	203
Using namespaces	203
Building Tcl/Tk Extensions under Windows 95/NT	203
Running SWIG from Developer Studio	203
Using NMAKE	204
Primitive Tcl Interface	205
Functions	205
Global variables	205
Constants	206
Pointers	207
Structures	207
C++ Classes	208
The object-based interface	209
Creating new objects	209
Invoking member functions	210
Deleting objects	210
Accessing member data	211
Changing member data	211
Managing Object Ownership	211
Relationship with pointers	212
Examples	212
Accessing arrays	212
Exception handling	214
Typemaps	215
What is a typemap?	215
Tcl typemaps	216
Typemap variables	216
Name based type conversion	217
Converting a Tcl list to a char **	217
Remapping constants	218
Returning values in arguments	219
Mapping C structures into Tcl Lists	221
Useful functions	222
Standard typemaps	223
Pointer handling	223
Configuration management with SWIG	224
Writing a main program and Tcl AppInit()	224
Creating a new package initialization library	226
Combining Tcl/Tk Extensions	227
Limitations to this approach	228
Dynamic loading	228
Turning a SWIG module into a Tcl Package	229
Building new kinds of Tcl interfaces (in Tcl)	230
Shadow classes	231

SWIG and Java.....235

Preliminaries	235
Running SWIG	235
Additional Commandline Options	236

Table of Contents

SWIG and Java

<u>Getting the right header files.....</u>	236
<u>Compiling a dynamic module.....</u>	236
<u>Using your module.....</u>	237
<u>Compilation problems and compiling with C++.....</u>	237
<u>Building Java Extensions under Windows 95/NT.....</u>	237
<u>Running SWIG from Developer Studio.....</u>	237
<u>Using NMAKE.....</u>	238
<u>The low-level Java/C interface.....</u>	239
<u>Modules and the module class.....</u>	239
<u>Functions.....</u>	239
<u>Variable Linking.....</u>	240
<u>Enums.....</u>	240
<u>Constants.....</u>	241
<u>Pointers.....</u>	241
<u>Structures.....</u>	242
<u>C++ Classes.....</u>	242
<u>Java shadow classes.....</u>	243
<u>A simple example.....</u>	244
<u>Why write shadow classes in Java?.....</u>	245
<u>Automated shadow class generation.....</u>	245
<u>Compiling modules with shadow classes.....</u>	246
<u>Where to go for more information.....</u>	246
<u>Examples.....</u>	246
<u>Exception handling.....</u>	246
<u>Remapping C datatypes with typemaps.....</u>	248
<u>Default type mapping.....</u>	248
<u>What is a typemap?.....</u>	249
<u>Java typemaps.....</u>	250
<u>Typemap variables.....</u>	251
<u>Typemaps for C and C++.....</u>	251
<u>Name based type conversion.....</u>	252
<u>Converting Java String arrays to char **.....</u>	252
<u>Using typemaps to return arguments.....</u>	255
<u>Accessing array structure members.....</u>	256
<u>Pointer handling.....</u>	257
<u>The gory details of shadow classes.....</u>	257
<u>A simple shadow class.....</u>	257
<u>Generated class.....</u>	258
<u>The this pointer.....</u>	259
<u>Object ownership.....</u>	259
<u>Constructors and destructors.....</u>	259
<u>Member data.....</u>	260
<u>Shadow Functions.....</u>	260
<u>Shadow class pointer handling.....</u>	261
<u>Methods that return new objects.....</u>	262
<u>Global variables and functions.....</u>	263
<u>Nested objects.....</u>	263
<u>Inheritance and shadow classes.....</u>	265
<u>Shadow classes and garbage collection.....</u>	267
<u>Performance concerns and hints.....</u>	268
<u>Java pragmas.....</u>	268

Table of Contents

SWIG and Java

<u>Deprecated pragmas</u>	270
<u>Pragma uses</u>	270
<u>Dynamic linking problems</u>	271
<u>Tips</u>	271
<u>Known bugs</u>	271
<u>G.1 Meaning of "Module"</u>	273
<u>G.2 Linkage</u>	273
<u>G.3 Underscore Folding</u>	273
<u>G.4 Typemaps</u>	275
<u>G.5 Smobs</u>	276
<u>G.6 Exception Handling</u>	276
<u>G.7 Procedure documentation</u>	277
<u>G.8 Procedures with setters</u>	277

G SWIG and Guile.....278

SWIG and Ruby.....279

<u>Preliminaries</u>	279
<u>Running SWIG</u>	279
<u>Compiling a dynamic module</u>	280
<u>Using your module</u>	280
<u>Building Ruby Extensions under Windows 95/NT</u>	280
<u>Running SWIG from Developer Studio</u>	280

SWIG and PHP4.....283

<u>Preliminaries</u>	283
<u>Building PHP4 Extensions</u>	283
<u>Building a loadable extension</u>	284
<u>Basic PHP4 interface</u>	284
<u>Functions</u>	284
<u>Global Variables</u>	285
<u>Pointers</u>	285
<u>Structures and C++ classes</u>	285
<u>Constants</u>	286
<u>Shadow classes</u>	287
<u>Constructors and Destructors</u>	288
<u>Static Member Variables</u>	288
<u>PHP4 Pragmas</u>	289
<u>Building extensions into php</u>	289
<u>To be furthered...</u>	290

Extending SWIG.....291

<u>Introduction</u>	291
<u>Prerequisites</u>	291
<u>High Level Overview</u>	291

11 Advanced Topics.....305

<u>Creating multi-module packages</u>	305
<u>Runtime support (and potential problems)</u>	305
<u>Why doesn't C++ inheritance work between modules?</u>	305
<u>The SWIG runtime library</u>	306

Table of Contents

11 Advanced Topics

<u>A few dynamic loading gotchas</u>	308
<u>Dynamic Loading of C++ modules</u>	309
<u>Inside the SWIG type-checker</u>	309
<u>Type equivalence</u>	310
<u>Type casting</u>	311
<u>Why a name based approach?</u>	311
<u>Performance of the type-checker</u>	312
<u>SWIG Users Manual</u>	312

SWIG1.3 Development Documentation

Last update : SWIG-1.3.11 (31 January 2002)

Authors:

- David Beazley (beazley@cs.uchicago.edu)
- William Fulton (wsf@fultondesigns.co.uk)
- Matthias Köppe (mkoeppe@mail.math.uni-magdeburg.de)
- Lyle Johnson (ljohnson@resgen.com)
- Richard Palmer (richard@magicality.org)

The SWIG documentation is currently being updated to reflect new SWIG features and enhancements. This update process is currently unfinished. There is a lot of old documentation and it's going to take some time to update all of it. Please bear with us (or volunteer to help!).

Documentation updated to SWIG-1.3

- [Preface](#)
- [Introduction](#)
- [Getting started on Windows](#)
- [Scripting](#)
- [SWIG Basics](#) (Read this!)
- [SWIG and C++](#) (New!)
- [The SWIG Library](#)
- [Typemaps](#) (Newly updated!)
- [Exception handling](#) (Newly updated!)
- [Java support](#)
- [Guile support](#)
- [Ruby support](#) (very minimal documentation)
- [PHP support](#) (in progress)

Documentation partially updated to SWIG-1.3

This documentation is in the process of being updated. Some information is missing or incorrect.

- [Tcl support](#)
- [Python support](#)
- [Extending SWIG](#)

Documentation that has not yet been updated

This documentation has not been updated, 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.

- [Documentation system](#) (empty)
- [Perl5 support](#) (some cleanup)

- [Advanced topics](#)

Documentation not yet written

- Mzscheme module

Preface

Introduction

SWIG 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.

Special Introduction for Version 1.3

Since SWIG was released in 1996, its user base and applicability has continued to grow. Although its development has gone through several stages of inactivity (mostly due to Dave's thrashing as a new assistant professor), development has continued. Today, an active effort is underway to redevelop many parts of the system and to add new language support such as Ruby, Guile, and Java.

Currently, development of SWIG is managed at the University of Chicago with assistance from SourceForge.

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-1.4 release.

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://mailman.cs.uchicago.edu/listinfo/swig>

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>

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.

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 most of the language module chapters are still out of date.

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.

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, and Harco de Hilster.

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.

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 swig-dev@cs.uchicago.edu or report a bug at <http://www.swig.org>. 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.

SWIG 1.3 – Last Modified : December 9, 2001

1 Introduction

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.

Why use SWIG?

Although C is great for high-performance number crunching and systems programming, trying to make an interactive and highly flexible C program is a pain. Even though it is possible to build a user interface using command line options, a home grown command interpreter, or a graphical user interface, this often results in a program that is hard to extend, hard to modify, hard to port between platforms, and hard to use. Furthermore, this sort of activity wastes a lot of development time because it usually diverts everyone's attention away from the real problem that they're trying to solve.

Many of the problems with C are due to the way in which many programs are organized. For example, a lot of programs are structured as follows:

- 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).

In this case, the `main()` program may read command line options or simple commands from `stdin`. However, modifying or extending the program to do something new requires changing the C code, recompiling, and testing. If you make a mistake, you need to repeat this cycle until things work. Of course, as more and more features are added, the program usually turns into a horrible mess that is even more difficult to modify than before (although

1 Introduction

this undoubtedly increases the job security of the programmer).

A common mistake is to assume that all of the problems with C can somehow be fixed by using a better C—perhaps an undebuggable language with unreadable syntax, complicated semantics, and nearly infinite compilation time. This is an unfortunate.

Perhaps a better approach is to place your application under the control of a very high-level language such as a common scripting language interpreter. High level languages excel at turning hard problems into easy tasks. They also provide a nice framework for managing software components and gluing different systems together. Not only that, they make it easy for users to configure the software to their liking and to program it to perform new tasks without ever having to touch a C/C++ compiler.

SWIG simplifies the task of incorporating C++ code into a high-level programming environment. Specifically, rather than creating a huge monolithic package, SWIG allows you to restructure your application as a collection of functions and variables that can be accessed from the convenience of a high-level language. With this model, all of the functionality of your C program is retained. The only difference is that the high-level program logic and control is now driven by the high-level language instead of a low level `main()` function.

SWIG tries to make the integration between scripting languages and C as painless as possible. This allows you to focus on the underlying C program and using the high-level scripting language interface, but not the tedious and complex chore of making the two languages talk to each other.

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) :

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.

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
Generating wrappers for Tcl.
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.

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
Generating wrappers for Perl5
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
unix >
```

Building a Python module

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

1 Introduction

```
unix > swig -python example.i
Generating wrappers for Python
unix > gcc -c -fpic example.c example_wrap.c -I/usr/local/include/python2.0
unix > gcc -shared example.o example_wrap.o -o examplemodule.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
```

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 source and specifying a module name as follows

```
% swig -perl5 -module example example.c
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
```

Of course, there are some restrictions as SWIG is not a full C/C++ parser. If you make heavy use of the C preprocessor, complicated declarations, or C++, giving SWIG a raw source file probably isn't going to work very well (in this case, you would probably want to use a separate interface file).

SWIG provides its own version of the C preprocessor. Thus, if you want to make a combination SWIG/C header file, you might write the following:

```
/* File : example.h */
#ifdef SWIG
%module example
#include tclsh.i
#endif
extern double My_variable;
extern int fact(int);
extern int my_mod(int n, int m);
```

Building libraries and modules

In addition to generating wrapper code, SWIG provides extensive support for handling multiple files and building

interface libraries. For example, our `example.i` file, could be used in another interface as follows :

```
%module foo
#include example.i           // Get definitions from example.i

... Now more declarations ...
```

In a large system, an interface might be built from a variety of pieces like this :

```
%module package

#include network.i
#include file.i
#include graphics.i
#include objects.i
#include simulation.i
```

SWIG comes with a library of existing functions known as the SWIG library. The library contains a mix of language independent and language dependent functionality. For example, the file ``array.i'` provides access to C arrays while the file ``wish.i'` includes specialized code for rebuilding the Tcl wish interpreter. Using the library, you can use existing modules to build up your own personalized environment for building interfaces. If changes are made to any of the components, they will appear automatically the next time SWIG is run.

C syntax, but not a C compiler

SWIG uses ANSI C syntax, but is not a full ANSI C compiler. By using C syntax, SWIG is relatively easy to use with most C programs. This avoids the problems with other interface generation tools that require programs to formally specify an interface in a special purpose interface definition language. On the other hand, using C syntax can be ambiguous. For example, if you have the following declaration,

```
int foo(double *a);
```

SWIG doesn't know if `a` is an array of some fixed size, a dynamically allocated block of memory, a single value, or an output value of the function. For the most part, SWIG takes a literal approach (`a` is obviously a `double *`) and leaves it up to the user to use the function in a correct manner. Fortunately, there are several ways to help SWIG out using additional directives and "helper" functions. Thus, the input to SWIG is often a mix of C declarations, special directives and hints. Like Perl, there is almost always more than one way to do almost anything.

SWIG does not currently parse every conceivable type of C declaration that it might encounter in a C/C++ file. Many things may be difficult or impossible to integrate with a scripting language. Thus, SWIG may not recognize certain advanced C/C++ constructs. Of course, SWIG's parser is always being improved so currently unsupported features may be supported in later releases.

Non-intrusive interface building

When used as intended, SWIG requires minimal modification to existing C code. This makes SWIG extremely easy to use with existing packages and promotes software reuse and modularity. By making the 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.

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). Ideally, SWIG should be invoked automatically inside a Makefile just as one would call the C compiler. You should think of your scripting 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.

SWIG 1.3 – Last Modified : November 28, 2001

Getting started on Windows

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 order to run some examples using Visual C++.

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 the [supplied instructions](#).

SWIG Windows Examples

Using Microsoft Visual C++ is the most successful 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. These project files have been set up to use SWIG in a custom build rule for the SWIG interface (.i) file.

More information on each of the examples is available with the examples on the [SWIG Examples](#) page.

Note that no SWIG language runtime libraries have been supplied. The examples which have a Microsoft Visual C++ project file do not need the runtime libraries. In fact the vast majority of the examples do not need the SWIG runtime libraries.

Instructions for using the Examples with Visual C++

Ensure the SWIG executable is as supplied in the SWIG root directory in order for the examples to work. Each language requires 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 the .dsp file, Visual C++ will create a workspace for you (.dsw file). 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 have a look at the section on building extensions for your chosen language module.

Python

PYTHON_INCLUDE : Set this to the directory that contains python.h

PYTHON_LIB : Set this to the python library including path for linking with

Getting started on Windows

Example using Python 2.1.1:

```
PYTHON_INCLUDE: d:\python21\include  
PYTHON_LIB: d:\python21\libs\python21.lib
```

TCL

TCL_INCLUDE : Set this to the directory containing tcl.h

TCL_LIB : Set this to the TCL library including path for linking with

Example using ActiveTcl 8.3.3.3

```
TCL_INCLUDE: d:\tcl\include  
TCL_LIB: d:\tcl\lib\tcl83.lib
```

Perl

PERL5_INCLUDE : Set this to the directory containing perl.h and perl.lib

Example using nsPerl 5.004_04:

```
PERL5_INCLUDE: D:\nsPerl5.004_04\lib\CORE
```

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
```

Ruby

RUBY_INCLUDE : Set this to the directory containing ruby.h

RUBY_LIB : Set this to the ruby library including path for linking with

Example using Ruby 1.6.4:

```
RUBY_INCLUDE: D:\ruby\lib\ruby\1.6\i586-mswin32  
RUBY_LIB: D:\ruby\lib\mswin32-ruby16.lib
```

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 could try using Cygwin as described in the following section.

SWIG on Cygwin and Mingw

SWIG can also be compiled and run using [Cygwin](#) 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 and many of the modules do not work under Cygwin due to difficulties in building Dynamic Link Libraries (dlls). Currently only the Java module is known to work, some of the others may or may not.

Building swig.exe on Windows

If you want to replicate the build of swig.exe that comes with the download, follow the following instructions. 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.

Building swig.exe using Cygwin and Mingw

- Install [Cygwin](#)
- Install [Mingw](#)
- Ensure that the mingw bin directory is before the cygwin bin directory in your path.
- Follow the usual Unix instructions in the README file in the SWIG root directory to build swig.exe.

Try running `./autogen` from the SWIG root directory before running `./configure` in order to use the latest autoconf and automake tools.

Note that SWIG can be built using just cygwin, ie no mingw installed. However, the SWIG executable will then require the cygwin DLL.

Building swig.exe alternatives

If you don't want to install cygwin and 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.

Running the examples on Windows using Cygwin and Mingw

Follow the Unix instructions in the README file in the SWIG root directory to build the examples. Note that a

```
make -k check
```

command is not very successful on Cygwin/mingw. Some modules require the Makefiles setting up by hand in order to get the dlltool to work correctly. The recommended way is to use Visual C++ to compile the output from swig.exe.

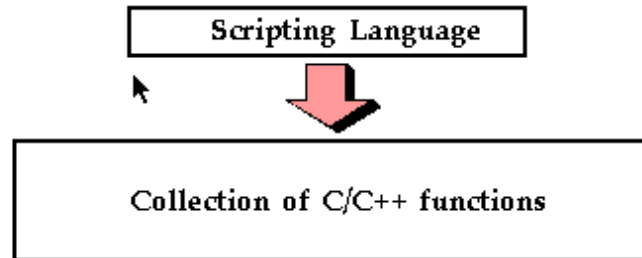
Try running `./autogen` from the SWIG root directory before running `./configure` in order to use the latest autoconf and automake tools.

2 Scripting Languages

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

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.

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.

Wrapper functions

Suppose you have an ordinary C function like this :

```
int fact(int n) {
```

2 Scripting Languages

```
    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.

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.

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.

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]
```

2 Scripting Languages

```
% 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`.

Shadow classes

In certain cases, it is possible to use the low-level accessor functions to create something known as a "shadow" class. A "shadow 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 "shadows" the real C++ class). For example, if you have the following C definition :

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

A shadow 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 shadow 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.

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.

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
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
```

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.

2 Scripting Languages

- 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.

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.

SWIG 1.3 – Last Modified : July 16, 2001

3 SWIG Basics

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.

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

-tcl           Generate Tcl wrappers
-perl          Generate Perl5 wrappers
-python       Generate Python wrappers
-guile        Generate Guile wrappers
-ruby         Generate Ruby wrappers
-java         Generate Java wrappers
-mzscheme     Generate mzscheme wrappers
-php          Generate PHP wrappers
-c++          Enable C++ parsing
-Idir         Add a directory to the file include path
-lfile        Include a SWIG library file.
-c           Generate raw wrapper code (omit supporting code)
-o outfile    Name of output file
-module name  Set the name of the SWIG module
-Dsymbol      Define a preprocessor symbol
-version      Show SWIG version number
-swiglib      Show location of SWIG library
-help         Display all options
```

Additional options are often defined for each target language. A full list can be obtained by typing `swig -help` or `swig -lang -help`.

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

3 SWIG Basics

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 to the resulting output file. The enclosed text is not parsed or interpreted by SWIG. Although the use of a `%{ , %}` block is optional, most interface files have one to include header files and other supporting C declarations. 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.

SWIG Output

The output of SWIG is a C/C++ file that contains all of the wrapper code needed to build an extension module. 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 output file created by SWIG normally contains everything that is needed to construct a extension module for the target scripting language. SWIG is not a stub compiler nor is 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.

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.

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" section near the end of this chapter.

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.

Parser Limitations

Although SWIG can parse most common 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;
/* Function declaration with unnecessary grouping */
int (foo)(int,int);
...
```

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

- 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 does not like 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 namespaces and 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

3 SWIG Basics

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).

It should also be noted that the SWIG parser was never really developed with the intent that it would be blindly used on raw C/C++ source code. Although parsing has become a lot more powerful in recent versions, the underlying assumption was that one would usually start with a header and enhance it by adding additional support code, cutting certain features out, supplying special SWIG directives, and so forth.

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.

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 since `long long` support is not part of the ANSI C standard, it is not universally supported by all C compilers. Make sure you are using a compiler that supports `long long` before trying to 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.

3 SWIG Basics

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.

Global Variables

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

```
%module example
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.

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 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.

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

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

Pointers and complex objects

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

Simple pointers

Pointers to primitive C datatypes such as

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

are fully supported by SWIG. SWIG encodes pointers 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 manipulate 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 above 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 would probably be best avoided.
- 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.

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).

3 SWIG Basics

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.

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.

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`.

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

3 SWIG Basics

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 *`.

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.

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.

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.

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.

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);
}
```

3 SWIG Basics

```
/* 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 `%readonly` 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.

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. 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] */
```

3 SWIG Basics

```
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.

Creating read-only variables

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

```
// File : interface.i

int      a;                      // Can read/write
%readonly
int      b,c,d                  // Read only variables
%readwrite
double   x,y                    // read/write
```

The %readonly directive enables read-only mode until it is explicitly disabled using the %readwrite directive.

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 */
```

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. This is described 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.

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++.

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 "", 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 "", 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.

3 SWIG Basics

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.

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.

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.

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;
}

```

3 SWIG Basics

```
    }

    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.

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.

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
```

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.

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

3 SWIG Basics

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

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

Finally, %addmethods 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;
    %addmethods {
        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 %addmethods 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
```

```

%addmethods 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 {
    %addmethods {
        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 `%addmethods` 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).

Nested structures

Occasionally, a C program will involve structures like this :

```

typedef struct Object {
    int objtype;
    union {
        int    ival;
        double dval;
        char   *strval;
        void   *ptrval;
    } intRep;
}

```

3 SWIG Basics

```
} 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 shadow 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.

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 ",
                     return TCL_ERROR;
    result = (double ) (arg1->x);
    Tcl_SetObjResult(interp,Tcl_NewDoubleObj((double) result));
    return TCL_OK;
}
```

SWIGTYPEP

The only exception to this rule are methods defined with `%addmethods`. 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.

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.

The output of SWIG

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

- **Runtime libraries.**

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.

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();
```

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.

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();
%}

```

SWIG Preprocessor

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 section describes some of these modifications.

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 auxiliary files.

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 advanced topic and is described in a later chapter.

3 SWIG Basics

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.

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
SWIGTCL	Defined when using Tcl
SWIGTCL8	Defined when using Tcl8.0
SWIGPERL	Defined when using Perl
SWIGPERL5	Defined when using Perl5
SWIGPYTHON	Defined when using Python
SWIGGUILE	Defined when using Guile
SWIGRUBY	Defined when using Ruby
SWIGJAVA	Defined when using Java
SWIGMZSCHEME	Defined when using Mzscheme
SWIGWIN	Defined when running SWIG under Windows
SWIGMAC	Defined when running SWIG on the Macintosh

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).

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.

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).

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
%}
```

3 SWIG Basics

the contents of the `{ ... }` block are copied without modification to the output (including all preprocessor directives).

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:

```
%addmethods 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:

```
%addmethods 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.

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.

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 swig-dev@cs.uchicago.edu or to the SWIG bug tracker on www.swig.org.

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.

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.

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
...
```

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.

How to avoid creating the interface from hell

SWIG makes it fairly easy to build a big interface really fast. In fact, if you apply it to a large enough package, you'll find yourself with a rather large amount of code being produced in the resulting wrapper file. For instance, wrapping a 1000 line C header file with a large number of structure declarations may result in a wrapper file containing 20,000–30,000 lines of code. Here are a few things you can do to make smaller interface:

- It is usually not necessary to wrap every single function in a package. Try to eliminate the unneeded ones.
- SWIG does not require structure definitions to operate. If you are never going to access the members of a structure, don't wrap the structure definition.
- Eliminate unneeded members of C++ classes.
- Think about what you are doing. If you are only using a subset of some library, there is no need to wrap the whole thing.
- Write support or helper functions to simplify common operations. Some C functions may not be easy to

use in a scripting language environment. You might consider writing an alternative version and wrapping that instead.

SWIG 1.3 – Last Modified : December 30, 2001

4 SWIG and C++

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++ is mostly an extension of ANSI C wrapping and that material will be useful in understanding this chapter.

Comments on C++ Wrapping

Because of its sheer complexity, and the fact that C++ can be difficult to integrate with itself let alone other languages, SWIG is only able to provide support for a subset of C++ features.

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, what you are really trying to do is 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 have to be careful.

To wrap C++, SWIG takes a deliberately conservative, low-level, and non-intrusive approach. For one, SWIG generates all of its C++ wrappers so that they have standard ANSI C linkage. This low-level interface is then used as a basis for building the resulting scripting language module (which may or may not utilize OO features). Second, SWIG does not encapsulate C++ classes inside special adaptor or proxy classes, it does not rely upon additional template magic, nor does it use C++ inheritance. 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.

Supported C++ features

SWIG currently supports the following C++ features :

- Simple classes.
- Constructors and destructors
- Virtual functions
- Public inheritance (including multiple inheritance)
- Static functions

4 SWIG and C++

- Function and method overloading (with renaming)
- Operator overloading for most standard operators
- References
- Simple templates
- Pointers to members

The following C++ features are not currently supported :

- Automatic function and method overloading
- Nested classes
- Namespaces
- Template specialization
- Overloaded versions of certain operators (new, delete, etc.)

SWIG's C++ support is always improving so some of these limitations may be lifted in future releases. However, we make no promises.

A simple C++ example

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);
};
```

When compiling 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.

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;
}
```



```
}
```

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.
- 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;
```

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 parsing 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.

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.

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`.

Member data

Member data is handled in exactly the same manner as for C structures. A pair of accessor functions will be 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 `%readonly` and `%readwrite` 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:
    ...
    %readonly
        int length;
    %readwrite
    ...
};
```

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

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++).

Enums and constants

Enumerations and constants placed in a class definition are mapped 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.

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);
}
```

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

```
class Bar {
public:
    double
};
```

Generates code like this:

```
double *Bar_spam(Bar *obj) {
    double = obj->spam();
    return
}
```

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.

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);
```

4 SWIG and C++

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 have a default constructor.

If `Vector` is defined as class in the interface, 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 will 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. However, SWIG sometimes applies the transformation when it's not needed (this would occur if a declaration involving a value was wrapped before SWIG was able to determine if a class had a default constructor or not).

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).

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().

4 SWIG and C++

Although the low-level C-like interface is functional, most language modules also produce a higher level OO interface using a technique known as shadow classing. This approach is described shortly and can be used to provide a more natural C++ interface.

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 old behavior results in huge amounts of replicated code for large class hierarchies and makes 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 shadow-classing 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.

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.

Wrapping Overloaded Functions and Methods

This section describes the problem of wrapping overloaded C++ functions and methods. This has long been a limitation of SWIG that has only recently been addressed (primarily because we couldn't quite figure out how to

do it without causing a head–explosion or serious reliability problems). However, in order to understand the reasoning behind the current solution, it is important to better understand the problem.

In C++, functions and methods can be overloaded by declaring them with different type signatures. For example:

```
void foo(int);
void foo(double);
void foo(Bar *b, Spam *s, int );
```

Later, when a call to function `foo()` is made, the determination of which function to invoke is made by looking at the types of the arguments. For example:

```
int x;
double y;
Bar *b;
Spam *s;
int z;
...
foo(x);           // Calls foo(int)
foo(y);           // Calls foo(double)
foo(b,s,z);       // Calls foo(Bar *, Spam *, int)
```

It is important to note that the selection of the overloaded method or function is made by the C++ compiler and occurs at **compile time**. It does not occur as your program runs.

Internal to the C++ compiler, overloaded functions are mapped to unique identifiers using a name–mangling technique where the arguments are used to create a unique type signature that is appended to the name. This produces three unique function names that might look like this:

```
void foo__Fi(int);
void foo__Fd(double);
void foo__FP3BarP4Spami(Bar *, Spam *, int);
```

Calls to `foo()` are then mapped to an appropriate version depending on the types of arguments passed.

The implementation of overloaded methods in C++ is difficult to translate directly to a scripting language environment because it relies on static type–checking and compile–time binding of methods—neither of which map to the dynamic environment of an interpreter. For example, in Python, Perl, and Tcl, it is simply impossible to define three entirely different versions of a function with exactly the same name within the same scope. The repeated definitions simply replace previous definitions.

Therefore, to solve the overloading problem, let's first look at several approaches that have been proposed as solutions, but which are **NOT** used to solve the overloading problem in SWIG.

- **Explicit renaming.** In earlier versions of SWIG, the only way to handle overloading was to explicitly rename overloaded methods to a unique name using the `%name` directive. For example:

```
void foo(int);
%name(foo_d) foo(double);
%name(foo_barspam) foo(Bar *, Spam *, int);
```

Although this certainly works, it is extremely annoying to explicitly annotate every class with a bunch of `%name` directives like that. In fact, it's so annoying that this really isn't a viable solution at all (except in cases where there is very little overloading). Dave sincerely apologizes for ever thinking that this

approach was good enough—however, let's try to forget the past and move on.

- **Name Mangling.** Another approach to overloading would be to automatically generate name-mangled versions of the functions. Although this would definitely work, it would also make the scripting language interface extremely annoying to use. For instance, does anyone really want to type things like this in their program?

```
foo__FP3BarP4Spami(b,s,i);
```

Needless to say, this approach is not used by SWIG nor has it ever been seriously considered.

- **Simplified Name Mangling.** An alternative name mangling approach would be to generate a simplified name mangling. For example, maybe you could just take the first letter of each type and use it as the signature. For example:

```
void foo(int);           // becomes foo_i(int)
void foo(double);        // becomes foo_d(int)
void foo(Bar *, Spam *, int); // becomes foo_BSi(int)
```

Although a lot more readable than the fully mangled version, this now has the problem of naming clashes. For instance, what is supposed to happen with these two functions?

```
void foo(int i);          // ?????
void foo(instance *obj);  // ?????
```

Also, what happens if the mangled version happens to match a legitimate identifier name used elsewhere in the program? One could use the %name directive to resolve such a conflict, but this tends to defeat the whole point. Although this might work in simple cases, there are still a number of obvious problems.

- **Numbering.** Another simple solution might be to number all of the overloaded methods in order of definition. For example:

```
void foo(int);           // becomes foo_1(int)
void foo(double);        // becomes foo_2(int)
void foo(Bar *, Spam *, int); // becomes foo_3(int)
```

Unfortunately, the numbering doesn't give any clues about what the actual function is. Also, if the order changes or a new function is added, all of the numbers might change—breaking all of the programs written against the interface. There is also a tiny problem of naming methods with multiple inheritance:

```
class X {
public:
    virtual void foo(int);           // X_foo_1
    virtual void foo(double);        // X_foo_2
};

class Y {
public:
    virtual void foo(long);           // Y_foo_1
    virtual void foo(Bar *, Spam *, int); // Y_foo_2
};

class Z : public X, public Y {
public:
    virtual void foo(double);          // Z_foo_1 ??? Mismatch X_foo_2
```



```

        virtual void foo(Bar *, Spam *, int); // Z_foo_2 ???
    // What happens to X_foo_1 and Y_foo_1 here?
};

```

In this case, the member functions have different names in the base class than they do in a derived class! Clearly this is just bizarre and not particularly obvious to someone who has to maintain the resulting code. Again, this doesn't seem to be a viable solution except in very simple cases.

- **Dynamic dispatch.** By far, the most powerful approach to the problem would be to implement some kind of dynamic dispatch mechanism in the code generator. For example, you might generate some code roughly equivalent to this pseudocode:

```

wrap_foo(args):
    if len(args) == 3:
        if (args[0].type == Bar and
            args[1].type == Spam and
            args[2].type == int):
            foo((Bar *) arg[0], (Spam *) arg[1], (int) arg[2])
    else if len(args) == 1:
        if args[0].type == int:
            foo((int) args[0])
        else if args[0].type == double:
            foo((double) args[0])
    else:
        raise "Bad arguments to foo"

```

Unfortunately there are serious problems with this approach as well. First, the addition of dynamic dispatch code introduces a performance hit on the execution time of overloaded methods since the arguments to each method call have to first be examined to figure out which function to dispatch. Although the sample code above doesn't look too bad, this procedure may involve interaction with the SWIG type-checker, typemaps (a SWIG customization scheme), and other more advanced parts of the interpreter. A more nasty problem has to do with functions that can accept the same type of scripting object. For example, if you have this,

```

void foo(int);
void foo(double);

```

the `foo(double)` function will probably accept both a scripting language integer and a floating point number as an argument. As a result, it's possible for the `foo(double)` function to hide the integer function `foo(int)` if arguments aren't checked in the correct order. For instance, if you switch the order of the two functions in the interface file, does `foo(int)` suddenly become unavailable? To deal with this problem, you might decide to make all of the overloaded functions additionally available through name mangling. However, that now introduces all of the problems of name mangling plus all of the problems of dynamic dispatch!

The bottom line is that even though some kind of dynamic dispatch scheme may be the "best" way to support overloading, it is difficult to implement and it has some serious shortcomings including performance, hiding of functions, and possibly poor interaction with some of SWIG's customization features.

- **Trial Execution.** An approach that is somewhat similar to dynamic dispatch would be to implement a trial execution scheme. In this case, each overloaded function would generate a unique wrapper function, possibly with a name-mangled name. For example, something like this:

4 SWIG and C++

```
wrap_foo_i(args) {
    ...
    foo((int) arg[0]);
    ...
}
wrap_foo_d(args) {
    ...
    foo((double) arg[0]);
    ...
}
wrap_foo_BSi(args) {
    ...
    foo((Bar *) arg[0], (Spam *) arg[1], (int) arg[2]);
    ...
}
```

Next, a top-level wrapper could be written like this:

```
wrap_foo(args) {
    if (wrap_foo_i(args) == SUCCESS) return SUCCESS;
    if (wrap_foo_d(args) == SUCCESS) return SUCCESS;
    if (wrap_foo_BSi(args) == SUCCESS) return SUCCESS;
    return ERROR, "No matching function foo";
}
```

Like dynamic dispatch, this solution suffers from a performance penalty from trying to start the execution of each possible function. In fact, the impact may be worse since the only way to determine the proper function is to try all possibilities until no errors occur (dynamic dispatch could make more intelligent choices). Another problem is that a function might throw an `ERROR` for a different reason than improper arguments (maybe the arguments were okay, but something happened during execution). Therefore, you would need to have some kind of special error condition to indicate an error in argument conversion. A more subtle problem arises with languages such as Ruby and Perl that handle errors by executing a `longjmp()` to return control back to the interpreter (in which case, the above approach won't work like we want). Finally, making this approach work with inheritance and all of SWIG's customization options is also problematic.

Of all of the schemes mentioned, trial execution is the most likely feature that might be added to SWIG in the future. However, no such support is planned at this time.

- **Don't allow overloading.** Easy to implement and extremely annoying to C++ programmers, but not particularly useful.

Alas, what to do about overloading?

Although it would be nice to support an advanced wrapping technique such as dynamic dispatch or trial execution, both of these techniques are difficult (if not impossible) to implement in a completely general manner that would work in all situations and with all combinations of SWIG customization features. Therefore, rather than generate wrappers that only work some of the time, SWIG takes a slightly different approach.

Starting with SWIG-1.3.7, a very simple enhancement has been added to the `%rename` directive to help disambiguate overloaded functions and methods. Normally, the `%rename` directive is used to rename a declaration everywhere in an interface file. For example, if you write this,

```
%rename(foo) bar;
```

all occurrences of "bar" will be renamed to "foo" (this feature was described a little earlier in this chapter in the section "Renaming Declarations"). By itself, this doesn't do anything to help fix overloaded methods. However, the `%rename` directive can now be parameterized as shown in this 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'
    ...
};
```

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"
```

When used in this simple form, the renaming is applied to all global functions and member functions that match the prototype. 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
```

4 SWIG and C++

```
...
};

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

Depending on your application, it may make more sense to include `%rename` specifications in the class definition. 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 swig-dev@cs.uchicago.edu).

Although the `%rename` approach does not automatically solve the overloading problem for you (you have to supply a name), SWIG's error messages have been improved to help. For example, consider this interface file:

```
%module foo

class Spam {
public:
    void foo(int);
    void foo(double);
    void foo(Bar *, Spam *, int);
};
```

If you run SWIG on this file, you will get the following error messages:

```
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 )
```

The error messages indicate the problematic functions along with their type signature. In addition, the previous definition is supplied. Therefore, you can just look at these errors and decide how you want to handle the overloaded functions. For example:

```
%module foo
%rename(foo_d)      Spam::foo(double);           // name foo_d
%rename(foo_barspam) Spam::foo(Bar *, Spam *, int); // name foo_barspam
...
class Spam {
...
};
```

And again, for a class hierarchy, you may be able to solve all of the problems by just renaming members in the base class—those renamings automatically propagate to all derived classes.

Another way to resolve overloaded methods is to simply eliminate conflicting definitions. An easy way to do this is to use the `%ignore` directive. `%ignore` works exactly like `%rename` except that it forces a declaration to disappear. 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`.

A few implementation notes about the enhanced `%rename` directive and `%ignore`:

- 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 are applied to both qualified and non-qualified members. For example, if you have a class like this:

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

the declaration `%rename(name) Foo::bar()` applies to the qualified member `bar() const`. However, an often overlooked C++ feature is that classes can define two different overloaded members that differ only in their qualifiers, like this:

```
class Foo {
public:
    ...
    void bar();           // Unqualified member
    void bar() const;     // Qualified member (OK)
    ...
};
```

Even when renaming is used, this still generates an error (both `bar()` methods will be renamed to the same thing). However, if you want to silence the errors, `%rename` and `%ignore` can be further specialized with qualifiers. For example, the following directive would tell SWIG to ignore the `const` version of `bar()` above:

```
%ignore Foo::bar() const;    // Ignore bar() const, but leave other bar() alone
```

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 Complex {
        rpart = c.rpart;
        ipart = c.ipart;
        return *this;
    }
    Complex operator+(const Complex const &c {
        return Complex(rpart+c.rpart, ipart+c.ipart);
    }
}
```

```

Complex operator-(const Complex const {
    return Complex(rpart-c.rpart, ipart-c.ipart);
}
Complex operator*(const Complex 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
```

4 SWIG and C++

```
%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.

Adding new methods

New methods can be added to a class using the `%addmethods` directive. This directive is primarily used in conjunction with shadow 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 ...
    %addmethods {
        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 `%addmethods` directive follows all of the same conventions as its use with C structures.

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);
```

Starting with SWIG-1.3.7, simple C++ template declarations can also be wrapped. 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 (and admittedly lame) template class declaration:

```
// File : list.h
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() {
```

4 SWIG and C++

```
        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.

To create wrappers for a specific template instantiation, use the `%template` directive like this:

```
/* 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. Most target languages do not recognize identifiers such as `List<int>`. Therefore, each instantiation of a template has to be associated with a nicely formatted identifier such as `intList` or `doubleList`. Furthermore, due to the details of the underlying implementation, the name you select has to be unused in both C++ and the target scripting language (e.g., the name must not match any existing C++ typename, class name, or declaration name).

Since most C++ compilers are nothing more than glorified preprocessors (sic) *and* C++ purists really hate macros, SWIG internally handles templates by converting them into macros and performing expansions using the preprocessor (well, actually it's somewhat more complicated than this, but the preprocessor is used for part of it). Specifically, the `%template(intList) List<int>` declaration results in a macro expansion that generates code roughly like this (which is then parsed to create the interface):

```
// Example of how templates are internally expanded by SWIG
%{
// Define a nice name for the instantiation
typedef List<int> intList;
%}
// Provide a simple class definition with types filled in
class intList {
private:
    int *data;
    int nitems;
    int maxitems;
public:
    intList(int max) {
        data = new int [max];
        nitems = 0;
        maxitems = max;
    }
    ~intList() {
        delete [] data;
    };
    void append(int obj) {
        if (nitems < maxitems) {
            data[nitems++] = obj;
        }
    }
    int length() {
        return nitems;
    }
    int get(int n) {
```

```

        return data[n];
    }
};

```

SWIG can also generate wrappers for function templates using a similar technique. For example:

```

// Function template
template 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.

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 will generate an warning message.

4 SWIG and C++

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)
...
```

If your goal is to make someone's head explode more than usual, SWIG directives such as `%rename` and `%addmethods` can be included directly in template definitions. Not only that, since SWIG has the advantage of using the preprocessor for template expansion, standard C preprocessor operators such as `#` and `##` can be applied to template parameters (an obvious oversight of the C++ standard that SWIG now corrects). For example:

```
// File : list.h
template<class T> class List {
    ...
public:
    %rename(__getitem__) get(int);
    List(int max);
    ~List();
    ...
    T get(int index);
    %addmethods {
        char *__str__() {
            /* Make a string representation */
            ...
        }
        /* Return actual type of template instantiation as a string */
        char *ttype() {
            return #T;
        }
    }
};
```

In this example, the extra SWIG directives are propagated to *every* template instantiation.

In addition, the `%addmethods` directive can be used to add additional methods to a specific instantiation. For example:

```
%template(intList) List<int>;

%addmethods intList {
    void blah() {
        printf("Hey, I'm an intList!\n");
    }
};
```

Needless to say, SWIG's template support provides plenty of opportunities to break the universe. That said, an important final point is that **SWIG performs no 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).

Finally, there are a few limitations in SWIG's current support for templates:

- Template specialization or partial specialization is not supported. For example:

```
class List<int> {
    ...
};
```

- Class templates may not be defined as the argument of another template. If you are doing this on purpose, we're all really impressed, but please stop.
- Member templates aren't supported.
- Template support in SWIG is still relatively immature. Although the current implementation is probably enough to handle simple stuff, wrapping something like the STL would probably be rough going. If you succeed at this, we'd definitely like to know about it! If you tried and failed, we would also like to get feedback so that we can try to address the problems you encountered.

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) =
```

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.

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

4 SWIG and C++

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.

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. In fact, many C++ books actively encourage using `const` whenever possible (perhaps in an effort to maximize correctness through annoyance or by observing that an uncompileable program always runs correctly).

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.

Where to go for more information

SWIG 1.3 – Last Modified : December 30, 2001

4 Multiple files and the SWIG library

For increased modularity and convenience, it is often useful to break an interface specification up into multiple files or modules. This chapter describes SWIG's support for library files.

The `%include` directive

The `%include` directive inserts code from another file into the current interface file. It is primarily used to build a package from a collection of smaller modules. For example :

```
// File : interface.i
%module package
%include "equations.i"
%include "graphics.i"
%include "fileio.i"
%include "data.i"
%include "network.c"
%include "../Include/user.h"
```

In this case, SWIG creates a single wrapper file for a module `package` that contains all of the included declarations. Repeated `%module` directives in other files are simply ignored.

The `%include` directive can process SWIG interface files, C header files, and C source files (provided they are sufficiently clean). When processing a C source file, SWIG automatically declares all functions it finds as "extern". Thus, use of a header file may not be required in this case. Running SWIG on C++ source files is not recommended due to parser limitations

The `%import` directive

The `%import` directive is used to gather declarations from modules that are wrapped in a separate module or from files that you don't want to wrap into the current interface. The primary purpose is to collect type information and information about base classes.

For example, if you wanted to extract some common typedef declarations from a header file, you might write this:

```
%module foo
%import "types.h"
...
```

Similarly, if you are working with multiple SWIG modules, you might write the following to pick up C++ base classes. For example:

```
%module foo
// Grab the base class
%import "base.i"

// Define a derived class
class Foo : public Base {
...
};
```

If a file included with `%import` contains a `%module` directive, it is sometimes used by the target language

4 Multiple files and the SWIG library

module to coordinate the operation of more than one SWIG dynamically loaded module. This is an advanced topic that is described in the chapter on the SWIG runtime libraries.

Including files on the command line

Like the C or C++ compiler, SWIG can also include library files on the command line using the `-l` option as shown

```
# Include a library file at compile time
% swig -tcl -lwish.i interface.i
```

This is particularly useful for debugging and building extensions to different kinds of languages. When libraries are specified in this manner, they are included after all of the declarations in `interface.i` have been wrapped. Thus, this does not work if you are trying to include common declarations, typemaps, and other files.

The SWIG library

SWIG comes with a library of functions that can be used to build up more complex interfaces. As you build up a collection of modules, you may also find yourself with a large number of interface files. Although the `%include` directive can be used to insert files, it also searches the files installed in the SWIG library (think of this as the SWIG equivalent of the C library). When you use `%include` or `%import`, SWIG searches for files in the following order:

- The current directory
- Directories specified with the `-I` option
- `./swig_lib`
- `/usr/local/lib/swig_lib` (or wherever you installed SWIG)

Within each directory, you can also create subdirectories for each target language. If found, SWIG will search these directories first, allowing the creation of language-specific implementations of a particular library file.

You can override the location of the SWIG library by setting the `SWIG_LIB` environment variable.

Library example

The SWIG library is really a repository of useful modules that can be used to build better interfaces. To use a library file, simply use the `%include` directive with the name of a library file. For example :

```
%module example

#include pointer.i                                // Grab the SWIG pointer library

// a+b --> c
extern double add(double a, double b, double *c);
```

In this example, we are including the SWIG pointer library that adds functions for manipulating C pointers. These added functions become part of your module that can be used as needed. For example, we can write a Tcl script like this that involves both the `add()` function and two functions from the `pointer.i` library :

```

set c [ptrcreate double 0]      ;# Create a double * for holding the result
add 4 3.5 $c                    ;# Call our C function
puts [ptrvalue $c]              ;# Print out the result

```

Creating Library Files

It is easy to create your own library files. To illustrate the process, we consider two different library files—one to build a new `tclsh` program, and one to add a few memory management functions.

`tclsh.i`

To build a new `tclsh` application, you need to supply a `Tcl_AppInit()` function. This can be done using the following SWIG interface file (simplified somewhat for clarity) :

```

// File : tclsh.i
%{
#if TCL_MAJOR_VERSION == 7 && TCL_MINOR_VERSION >= 4
int main(int argc, char **argv) {
    Tcl_Main(argc, argv, Tcl_AppInit);
    return(0);
}
#else
extern int main();
#endif
int Tcl_AppInit(Tcl_Interp *interp){
    int SWIG_init(Tcl_Interp *);

    if (Tcl_Init(interp) == TCL_ERROR)
        return TCL_ERROR;

    /* Now initialize our functions */

    if (SWIG_init(interp) == TCL_ERROR)
        return TCL_ERROR;

    return TCL_OK;
}
%}

```

In this case, the entire file consists of a single code block. This code will be inserted directly into the resulting wrapper file, providing us with the needed `Tcl_AppInit()` function.

`malloc.i`

Now suppose we wanted to write a file `malloc.i` that added a few memory management functions. We could do the following :

```

// File : malloc.i
%{
#include <malloc.h>
%}

typedef unsigned int size_t
void *malloc(size_t nbytes);
void *realloc(void *ptr, size_t nbytes);
void free(void *);

```

4 Multiple files and the SWIG library

In this case, we have a general purpose library that could be used whenever we needed access to the `malloc()` functions. Since this interface file is language independent, we can use it anywhere.

Placing the files in the library

Although both of our examples are SWIG interface files, they are quite different in functionality since `tclsh.i` would only work with Tcl while `malloc.i` would work with any of the target languages. Thus, you should put these files into the SWIG library as follows :

```
./swig_lib/malloc.i
./swig_lib/tcl/tclsh.i
```

When used in other interface files, this allows us to use `malloc.i` with any target language while `tclsh.i` will only be accessible if creating for wrappers for Tcl (ie. when creating a Perl5 module, SWIG will not look in the `tcl` subdirectory).

It should be noted that language specific libraries can mask general libraries. For example, if you wanted to make a Perl specific modification to `malloc.i`, you could make a special version and call it `./swig_lib/perl5/malloc.i`. When using Perl, you'd get this version, while all other target languages would use the general purpose version.

Working with library files

There are a variety of additional methods for working with files in the SWIG library described next.

Wrapping a library file

If you would like to wrap a file in the SWIG library, simply give SWIG the name of the appropriate library file on the command line. For example :

```
unix > swig -python pointer.i
```

If the file `pointer.i` is not in the current directory, SWIG will look it up in the library, generate wrapper code, and place the output in the current directory. This technique can be used to quickly make a module out of a library file regardless of where you are working.

Checking out library files

At times, it is useful to check a file out of the library and copy it into the working directory. This allows you to modify the file or to simply retrieve useful files. To check a file out of the library, run SWIG as follows :

```
unix > swig -co -python array.i
array.i checked out from the SWIG library
unix >
```

The library file will be placed in the current directory unless a file with the same name already exists (in which case nothing is done).

The SWIG library is not restricted to interface files. Suppose you had a Perl script that you liked to use a lot. You could place this in the SWIG library. Now whenever you wanted to use it, you could retrieve it by issuing :

```
unix > swig -perl5 -co myscript.pl  
myscript.pl checked out from the SWIG library
```

Similarly, the library also contains Makefiles to build various extensions. For example, if you need a quick Makefile for building Tcl extension, type the following:

```
unix> swig -tcl -co Makefile  
Makefile checked out from the SWIG library
```

During installation, SWIG creates a collection of preconfigured Makefiles for various scripting languages. If you need to make a new module, just check out one of these Makefiles, make a few changes, and you should be ready to compile and extension for your system.

SWIG 1.3 – Last Modified : August 18, 2001

5 Documentation System

The documentation system is under repair and disabled in SWIG1.3. It will return in a later release.

SWIG 1.3 – Last Modified : August 18, 2001

6 Types and Typemaps

Introduction

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 may want a `char **` to act like a list of strings instead of a bare pointer. In another case, you may want to tell SWIG that a parameter of `double *result` is the output value of a function. Similarly, you might want to map a datatype of `float[4]` into a 4 element tuple. This chapter describes some of the techniques that can be used to customize SWIG's type handling in order to handle these situations. In particular, details of the underlying type system and typemaps, an advanced customization feature, are presented.

Before jumping in, it should be emphasized that typemaps are an advanced customization feature that provide access to some of SWIG's internals and low-level code generator. Furthermore, typemaps are generally not required to build a simple interface when first starting out. Therefore, the material in this chapter will appeal more to users who have already built a few simple interfaces, but would like to do more.

The Problem

Suppose that you wanted to provide a scripting language wrapper around a function with the following prototype:

```
int foo(int argc, char *argv[]);
```

If you do nothing, SWIG produces a wrapper that expects to receive a pointer of type `char **` as the second argument. For example, if you try to use the function you might get an error like this:

```
>>> foo(3,["ale","lager","stout"])
Traceback (most recent call last):
  File "", line 1, in ?
TypeError: Type error. Expected _p_p_char
>>>
```

One way to fix this problem is to write a few assist functions to manufacture an object of the appropriate type. For example:

```
%inline %{
char **new_args(int maxarg) {
    return (char **) malloc(maxarg*sizeof(char *));
}
void del_args(char **args, int nargs) {
    while (--nargs > 0) {
        free(args[nargs]);
    }
    free(args);
}
void set_arg(char **args, int n, char *value) {
    args[n] = (char *) malloc(strlen(value)+1);
    strcpy(args[n],value);
}
%}
```

6 Types and Typemaps

Now in the scripting language:

```
>>> args = new_args(3)
>>> args
_000f4248_p_p_char
>>> set_arg(args,0,"ale")
>>> set_arg(args,1,"lager")
>>> set_arg(args,2,"stout")
>>> foo(3,args)
>>> del_args(args,3)
```

Needless to say, even though this works, it isn't the most user friendly interface. It would be much nicer if you could simply make a list of strings work like a `char **`. For example:

```
>>> foo(3, ["ale","lager","stout"])
```

An even better approach might implicitly set the `argc` parameter and allow the following:

```
>>> foo(["ale","lager","stout"])
```

Similar sorts of problems also arise when creating wrappers for small arrays, output values, and certain kinds of data structures.

One of the reasons why SWIG does not provide automatic support for mapping scripting language objects such as lists and associative arrays into C is that C declarations often do not provide enough semantic information for SWIG to know exactly how this should be done. For example, if you have a function like this,

```
void foo(double *x, double *y, double *r);
```

it's not obvious what the arguments are supposed to represent. Are they single values? Are they arrays? Is a result stored in one of the arguments?

Even in our earlier example, there are many possible interpretations of the `char *argv[]` argument. For example, is the array supposed to be NULL-terminated? Are the elements of the array modified by the underlying C function? Does the first element have any special meaning such as the name of a program or function?

The only thing that SWIG really knows about both of these cases is that the argument is some kind of pointer (and in fact, SWIG is perfectly happy to generate code that simply passes pointers around). Any further interpretation of the pointer's meaning requires a little more information.

Typemaps

One way to provide more information about a particular C datatype is to attach a special code generation rule to the type known as a *typemap*. Typemaps are the primary customization mechanism used to modify SWIG's default type-handling behavior. For example, 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 source code, you need to give it additional information for the wrapper code to mimic this behavior. To do this, you can 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 typemap 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
>>>
```

6 Types and Typemaps

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
```

Managing input and output parameters

One of the most common applications of typemaps is to handle pointers that correspond to simple input, output, or mutable function parameters. Typically this problem arises when working with functions that return more than one value such as a function that returns both a result and a status code to indicate success. The `typemaps.i` file contains a variety of rules for managing such pointers to the primitive C datatypes.

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)
```

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)
```

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:

6 Types and Typemaps

```
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.

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.

Array handling

Write me.

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.

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.

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).

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 *;
```

Writing new typemaps

So far, only a few examples of *using* typemaps have been presented. However, if you're willing to get your hands dirty and dig into the internals of your favorite scripting language (and SWIG), it is possible to do much more.

Before diving in, it needs to be stressed that under normal conditions, SWIG does **NOT** require users to write new typemaps (and even when they are used, it is probably better to use them sparingly). A common confusion among some new users to SWIG is that they somehow need to write typemaps to handle new types when in fact they really only need to use a `typedef` declaration. For example, if you have a declaration like this,

```
void blah(size_t len);
```

you really only need to supply an appropriate `typedef` to make it work. For example:

```
typedef unsigned long size_t;
void blah(size_t len);
```

Typemaps are only used if you want to change the way that SWIG actually generates its wrapper code. For example, if you needed to express `size_t` as a string of roman numerals to preserve backwards compatibility with some piece of legacy software. A more practical application is the conversion of common scripting language objects such as lists and associative arrays into C datatypes. For example, converting a list of strings into a `char * []` as shown in the first part of this chapter.

Before proceeding, you should first ask yourself if it is really necessary to change SWIG's default behavior. Next, you need to be aware that writing a typemap from scratch usually requires a detailed knowledge of the internal C API of the target language. Finally, it should also be stressed that by writing typemaps, it is easy to break all of the output code generated by SWIG. With these risks in mind, this section describes the basics of the SWIG type system and typemap construction. Language specific information is contained in later chapters.

The SWIG type system

Typemaps are tightly integrated with the internal operation of the SWIG type system. Internal to SWIG, all C++ datatypes are managed as a pair of types (`type`, `ltype`). `type` is the actual C++ datatype as it is specified in the interface file. `ltype` is a modified version of the datatype that can be used as an assignable local variable (a type that can be used on the left-hand side of a C assignment operator). The relationship between these two types directly pertains to the generation of wrapper code. Specifically, `ltype` is used to declare local variables used during argument conversion whereas `type` is used to make sure the actual C/C++ function is called without any type-errors. For example, if you have a C declaration like this:

```
void func(..., type, ...);
```

The corresponding wrapper code will look approximately like this:

```
wrap_func(args) {
    ...
    ltype argn;                // Local type for argument
    ...
    argn = ConvertValue(args[n]);
    ...
    func(..., (type) argn, ...); // Cast back to type
    ...
}
```


The relationship between the real C++ datatype and its ltype value is determined by the following rules:

- All qualifiers (`const`, `volatile`, etc.) are stripped.
- C++ References are converted to pointers.
- Arrays are converted to a pointer to an array of one less dimension.
- Enumeration types are converted to an `int`.

For example:

type	ltype
object	object
object *	object *
const object *	object *
const object * const	object *
object	object *
object [10]	object *
object [10][20]	object (*)[20]

In certain cases, names defined with `typedef` are also expanded. For example, if you have a type defined by a `typedef` as follows:

```
typedef double Matrix[4][4];
```

the ltype of `Matrix` is set to `double (*)[4]` since there is no way for SWIG to create an assignable variable using variations of the `Matrix` typename.

It should be stressed that these rules also define the behavior of the SWIG run-time type checker. Specifically, all of the type checking described in Chapter 3 is actually performed using ltype values and not the actual C datatype. This explains why, for instance, there is no difference between pointers, references, and one-dimensional arrays when they are used in the corresponding scripting language module. It also explains why qualifiers never appear in the mangled type-names used for type checking.

What is a typemap?

A typemap is a code generation rule that is attached to a specific datatype. It is specified using the `%typemap` directive. For example, a simple typemap might look like this:

```
%module example
%typemap(in) int {
    $1 = (int) PyInt_AsLong($input);
    if (PyErr_Occurred()) return NULL;
    printf("received %d\n", $1);
}

int add(int a, int b);
```

In this case, the typemap defines a rule for handling input arguments in Python. When used in a Python script, you would get the following output:

```
>>> a = add(7,13)
received 7
received 13
```

6 Types and Typemaps

In the typemap specification, the symbols `$1` and `$input` are place holders for C variable names in the generated wrapper code. In this case, `$input` is a variable containing the raw Python object supplied as input and `$1` is the variable used to hold the converted value (in this case, `$1` will be a C local variable of type `int`). The `$1` variable is *always* the ltype of the type supplied to the `%typemap` directive.

To support different code-generation tasks, a variety of different typemaps can be defined. For example, the following typemap specifies how to result integers back to Python

```
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

In this case, `$result` refers to the Python object being returned to the interpreter and `$1` refers to the variable holding the raw integer value being returned.

At first, the use of `$1` to refer to the local variable in a typemap may be counterintuitive. However, this notation is used to support an advanced feature of typemaps that allow rules to be written for groups of consecutive types. For example, a typemap can be written as follows:

```
%typemap(in) (char *buffer, int len) {
    $1 = PyString_AsString($input);
    $2 = PyString_Size($input);
}
...

// Some function
int count(char *buffer, int len, char *pattern);
```

In this case, the pattern `(char *buffer, int len)` is handled as a single object. Within the typemap code, `$1` and `$2` are used to refer to each part of the pattern. For instance, `$1` is a variable of type `char *` and `$2` is a variable of type `int` (for the the curious, this naming scheme is roughly taken from yacc). The use of multi-argument maps is an advanced topic and is described a little later.

Compatibility note: Prior to SWIG1.3.10, typemaps used the special variables `$source` and `$target` to refer to local variables used during conversion. Unfortunately, the roles of these variables was somewhat inconsistent (and in some places their meaning switched depending on the type of the typemap). In addition, this naming scheme is awkward when extended to multiple arguments. Although `$source` and `$target` are still supported for backwards compatibility, all future use is deprecated and may be eliminated in a future release. In new versions, the local variables corresponding to the types in the typemap specification are referenced using `$1`, `$2`, and so forth. Objects in the target language passed as input are referenced by `$input`. Objects returned to the target language are referenced by `$result`.

Creating a new typemap

A new typemap is specified as follows :

```
%typemap(method) type {
    ... Conversion code ...
}

%typemap(method) type "Conversion code";

%typemap(method) type %{
    ... Conversion code ...
}
```

```
%}
```

method defines a particular conversion method and `type` is the actual C++ datatype as it appears in the interface file (it is not the `ltype` value described in the section on the SWIG type system). The code attached to the typemap is supplied in braces, a quoted string, or in a `%{ ... %}` block after the typemap declaration. If braces are used, they are included in the output—meaning that new local variables can be declared inside the typemap code (the typemap defines a new C block scope). Otherwise, the code is inlined into the generated wrappers with no surrounding braces.

Since typemap conversion code is almost always dependent on the target language, it is fairly common to surround typemap specifications with conditional compilation—especially if a module is being designed for use with multiple target languages. For example:

```
#ifdef SWIGPYTHON
%typemap(in) int {
    $1 = (int) PyInt_AsLong($input);
}
...
#endif
#ifdef SWIGPERL5
%typemap(in) int {
    $1 = (int) SvIV($input);
}
#endif
```

A single typemap rule can be applied to a list of matching datatypes by using a comma separated list. For example :

```
%typemap(in) int, short, long, signed char {
    $1 = ($1_ltype) PyInt_AsLong($input);
    if (PyErr_Occurred()) return NULL;
    printf("received %d\n", $1);
}
```

Here, `$1_ltype` is expanded into the local datatype used during code generation (this is the assignable version of the type described in the SWIG type system section). This form of specifying a typemap is a useful way to reduce the amount of typing required when the same typemap code might apply to a whole set of similar datatypes. Also, note that this syntax is not the same as a multi-argument typemap.

Typemaps may also be attached to specific declarator names as in:

```
%typemap(in) char **argv {
    ... Turn an array into a char ** ...
}
```

A "named" typemap only applies to declarations that exactly match both the C datatype and the name. Thus the `char **argv` typemap will only be applied to function arguments that exactly match "`char **argv`". Although the name is usually the name of a parameter in a function/method declaration, certain typemaps are applied to return values (in which case, the name of the corresponding function or method is used).

Typemaps can also be specified for a sequence of consecutive types by enclosing the types in parentheses. For example:

6 Types and Typemaps

```
%typemap(in) (char *buffer, int len) {  
    ...  
}
```

In this case, the typemap only applies to function/method arguments where the argument pair `char *buffer, int len` appears.

Deleting a typemap

A typemap can be deleted by providing no conversion code. For example:

```
%typemap(method) type; // Deletes this typemap
```

Copying a typemap

A typemap can be copied using the following declaration :

```
%typemap(method) type = srctype; // Copies a typemap
```

This specifies that the typemap for `type` should be the same as the `srctype` typemap. Here is an example:

```
%typemap(in) long = int;  
  
// Copy a typemap with names  
%typemap(in) int_64 *output = long *output;  
  
// Copy a multi-argument typemap  
%typemap(in) (char *data, int size) = (char *buffer, int len);
```

Typemap matching rules

When datatypes are processed in an interface file, SWIG tries to match types and names to a typemap rule as follows:

- Typemaps that exactly match the type and name (a named typemap).
- Typemaps that exactly match the type only (an unnamed typemap).
- Typemaps that match the stripped type and name (the type is stripped of qualifiers).
- Typemaps that match the stripped type only.
- If the type is an array, try to match against typemaps defined with dimensions set to `ANY`.
- If the type is equivalent to another type via `typedef`, typemaps for the `typedef` type are applied.
- Otherwise, a match is made against a default type.

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 rules are applied:

```
typedef int Integer;  
  
%typemap(in) int *x {  
    ... typemap 1  
}  
  
%typemap(in) int * {  
    ... typemap 2
```

```

}

%typemap(in) Integer *x {
    ... typemap 3
}

void A(int *x);      // int *x rule (typemap 1)
void B(int *y);      // int * rule (typemap 2)
void C(Integer *);   // int * rule (via typedef) (typemap 2)
void D(Integer *x);  // Integer *x rule (typemap 3)
void E(const int *); // int * rule (const stripped) (typemap 2)

```

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

```

Compatibility note: SWIG1.1 applied a complex set of type-matching rules in which a typemap for `int *` would also match many different variations including `int int []`, and qualified variations. This feature is revoked in SWIG1.3. Typemaps must now exactly match the types and names used in the interface file.

Compatibility note: Starting in SWIG1.3, typemap matching follows `typedef` declarations if possible (as shown in the above example). This type of matching is only performed in one direction. For example, if you had `typedef int Integer` and then defined a typemap for `Integer`, that typemap would *not* be applied to the `int` datatype. Earlier versions of SWIG did not follow `typedef` declarations when matching typemaps. This feature has primarily been added to assist language modules that rely heavily on typemaps (e.g., a typemap for "int" now defines the default for integers regardless of what kind of typedef name is being used to actually refer to an integer in the source program).

Common typemap methods

When a typemap is specified, the supplied method identifies a specific aspect of wrapper code generation. For instance, the "in" method used earlier pertains to incoming argument conversion. The following list contains other commonly used methods that are supported by most language modules:

- `%typemap(in)`
Marshal function arguments from the target language to a C representation.
- `%typemap(out)`
Convert a function return value to an object in the target language.
- `%typemap(ret)`
Used to clean up the return value of a function. For example, if you needed to deallocate memory.

6 Types and Typemaps

- `%typemap(freearg)`
Used to clean up argument values. For example, if memory was allocated to hold an argument prior to calling a function.
- `%typemap(argout)`
Used to handle arguments that hold output values. This code is used to extract the output values and generate result objects in the target language.
- `%typemap(ignore)`
Ignore an argument.
- `%typemap(default)`
Set a default value for an argument (making the argument optional).
- `%typemap(arginit)`
Initializes an argument to have a specific value prior to argument conversion.
- `%typemap(check)`
Allows an argument to be checked for a valid value. For example, if you want to disallow a NULL pointer or constrain an argument value to positive numbers.
- `%typemap(memberin)`
Used to supply special code for setting structure and class data members. Most commonly used to set array members.
- `%typemap(globalin)`
Used to supply special code for setting global variables. Most commonly used to set array variables.

Each typemap is applied in a very specific location within the wrapper functions generated by SWIG. Specifically, the general form a wrapper function is as follows:

```
_wrap_foo() {  
  
    /* Marshal input values to C */  
    [ arginit typemaps ]  
    [ ignore typemaps ]  
    [ default typemaps ]  
    [ in typemaps ]  
    [ check typemaps ]  
  
    /* Call the actual C function */  
    call foo  
  
    /* Return result to target language */  
    [ out typemap ]  
    [ argout typemaps ]  
    [ freearg typemaps ]  
    [ ret typemap ]  
}
```

To illustrate, it is often useful to write a simple interface file with some typemaps and to take a look at the generated wrapper code. To illustrate, consider the following interface file (with typemaps):

```
%module example
```

```

%typemap(in) int {
    // "in" typemap. $1, $input
}
%typemap(out) int {
    // "out" typemap. $1, $result
}
%typemap(ignore) int ignored {
    // "ignore" typemap. $1
}
%typemap(check) int {
    // "check" typemap. $1
}
%typemap(arginit) int *out {
    // "arginit" typemap. $1
}
%typemap(argout) int *out {
    // "argout" typemap. $1, $result
}
%typemap(ret) int {
    // "ret" typemap. $1
}
%typemap(freearg) int *out {
    // "freearg" typemap. $1
}

int foo(int, int ignored, int *out);

```

Now, let's take a look at the resulting wrapper function (generated for Python, but it looks similar for other languages):

```

static PyObject *_wrap_foo(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    int arg0 ;
    int arg1 ;
    int *arg2 ;
    PyObject * obj0  = 0 ;
    PyObject * argo2 =0 ;
    int result ;

    {
        // "arginit" typemap. arg2
    }
    {
        // "ignore" typemap. arg1
    }
    if(!PyArg_ParseTuple(args,(char *)"OO:foo",&obj0,&argo2)) return NULL;
    {
        // "in" typemap. arg0, obj0
    }
    if ((SWIG_ConvertPtr(argo2,(void **) &arg2,SWIGTYPE_p_int,1)) == -1) return NULL;
    {
        // "check" typemap. arg0
    }
    {
        // "check" typemap. arg1
    }
    result = (int )foo(arg0,arg1,arg2);
    {
        // "out" typemap. result, resultobj
    }
}

```

6 Types and Typemaps

```
{
    // "argout" typemap. arg2, resultobj
}
{
    // "freearg" typemap. arg2
}
{
    // "ret" typemap. result
}
return resultobj;
}
```

In this example, you can see how different typemaps are used for different purposes. For example, the "arginit" and "ignore" typemaps appear first, and are used to initialize variables before anything else occurs. The "in" typemap is used to help convert arguments from the target language to C. The "check" typemap appears just prior to the actual function call and is used to validate arguments. After the function has been called, the "out" and "argout" typemaps are used to create an output values. Typically, "argout" appends its result to any result already set by the "out" typemap. The last two typemaps, "freearg" and "ret" are used to perform cleanup actions.

Within each typemap, the `$1` is always replaced by a C local variable corresponding to that type. For example, you can see how `$1` is replaced by `arg0`, `arg1`, and `arg2` depending on the argument in question. For typemaps related to returning a result, `$1` is set to the local variable holding the raw result of the function call (in this case, the variable `result`).

Writing typemap code

The conversion code supplied to a typemap needs to follow a few conventions described here.

Local scope

Typemap code is normally enclosed in braces when it is inserted into the resulting wrapper code (using C's block-scope). It is perfectly legal to declare local and static variables in a typemap. For example, you could write this:

```
%typemap(in) int {
    int temp;
    temp = (int) SvIV($input);
    $1 = temp;
}
```

In this case, the local variable `temp` only exists inside the typemap code itself. It does not affect other variables in the wrapper function and it does not matter whether or not other typemaps happen to use the same variable name. Note: that if you specify typemap code using a string or a `%{ ... %}` block, the typemap code is not enclosed in braces like this.

Creating local variables

Sometimes it is useful to declare a new local variable that exists in the scope of the entire wrapper function. This can be done by specifying a typemap with parameters as follows :

```
%typemap(in) int *INPUT(int temp) {
    temp = (int) PyInt_AsLong($input);
    $1 = &temp;
}
```



```
}
```

In this case, `temp` becomes a local variable in the scope of the entire wrapper function. For example:

```
wrap_foo() {
    int temp;    <--- Declaration of temp goes here
    ...

    /* Typemap code */
    {
        temp = (int) PyInt_AsLong(...);
        ...
    }
    ...
}
```

When you set `temp` to a value, it persists for the duration of the wrapper function and gets cleaned up automatically on exit. This is particularly useful when a typemap needs to create a temporary value, but doesn't want to rely on heap allocation.

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(int *INPUT, int *INPUT, int *INPUT);
```

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;
    int temp1;    /* Locals declared in the INPUT typemap */
    int temp2;
    int temp3;
    ...
    {
        temp1 = (int) PyInt_AsLong(...);
        arg1 = &temp1;
    }
    {
        temp2 = (int) PyInt_AsLong(...);
        arg2 = &temp2;
    }
    {
        temp3 = (int) PyInt_AsLong(...);
        arg3 = &temp3;
    }
    ...
}
```

Some typemaps do not recognize local variables (or they may simply not apply). At this time, only the "in", "argout", "default", and "ignore" typemaps support local variables (typemaps that apply to conversion of arguments).

6 Types and 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 =
}

%typemap(argout) int * {
    PyObject *o = PyInt_FromLong(temp$argsnum);
    ...
}
```

In this case, the `$argnum` 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.

Special variables

Within the code supplied to a typemap, a number of special variable substitutions are made. This table describes these substitutions:

Variable	Meaning
<code>\$n</code>	The C local variable corresponding to type <i>n</i> in the typemap declaration.
<code>\$input</code>	The input object for an argument in the target language. This is only available in typemaps related to argument conversion.
<code>\$result</code>	The output object being returned by a wrapper function.
<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>\$/td></code>	Real C datatype of type <i>n</i> with one pointer added.
<code>\$</code>	ltype of type <i>n</i> with one pointer added.
<code>\$e</code>	Mangled form of type <i>n</i> with one pointer added.
<code>\$iptor</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.

Variables such as `$` and `*$1_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 * ($*1_type temp) {
    temp = PyInt_AsLong($input);
    $1 =
}

```

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 `$1_ltype 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] {
    $1_ltype 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;
    ...
    (($1_ltype) temp)[i][j] = x;    /* set a value */
    ...
}

```

6 Types and Typemaps

Another approach, which only works for arrays is to use the `$1_basetype` substitution. For example:

```
%typemap(in) int [10][20] {
    $1_basetype temp[10][20];
    ...
    temp[i][j] = x;    /* set a value */
    ...
}
```

Typemap Parameters

Sometimes, a typemap may take additional parameters that are specific to a certain language module. These are specified in the `%typemap` directive using keyword parameters as follows:

```
%typemap(in, name="value", name="value", ...) ...
```

A common usage of this feature is to alter the behavior of documentation strings or usage information. For example:

```
%typemap(in, doc="4-tuple") int [4] {
    ...
}
```

The exact set of recognized parameters depends on the target language. Further details are covered in the documentation for each language module.

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++) {

```

6 Types and Typemaps

```
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.

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;
    }
}
```

6 Types and Typemaps

```
    }  
}
```

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.

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 scount, char *swords[]) { ... }
%typemap(in) (int wcount, char *words[]) { ... }

void search_words(int scount, 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 scount, 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");
    }
}
```

6 Types and Typemaps

```
        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).

The 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).

6 Types and Typemaps

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. Typemaps definitions are usually found in the SWIG library in a file such as `python.swg`, `tcl8.swg`, etc.

The run-time type checker

A critical part of SWIG's operation is that of its run-time type checker. 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 --> | -----|
```

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 **) 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 **) $l_descriptor)) == -1) return NULL;
}
```

The most critical part is the typemap is the use of the `$l_descriptor` special variable. When placed in a typemap, this is expanded into the `SWIGTYPE_*` type descriptor object above. As a general rule, you should always use `$l_descriptor` instead of trying to hard-code the type descriptor name directly.

There is another reason why you should always use the `$l_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.

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 **) descr) == -1)) {
        abort();
    }
    ...
}
```

6 Types and Typemaps

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.

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(ignore) int *OUTPUT (int temp) {
    $1 =
}
%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 =
}

%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;
```

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;
}
```

6 Types and Typemaps

```
%}  
  
%typemap(in) float [ANY] {  
    $1 = convert_float_array($input, $1_dim0);  
    if (!$1) return NULL;  
}  
%
```

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.

SWIG 1.3 – Last Modified : December 9, 2001

7 Exceptions, Features, and other Customizations

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.

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 `%except` 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.

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() {
```

7 Exceptions, Features, and other Customizations

```
        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.

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.

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 {};
```

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`. 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"
```

Applying exception handlers to specific datatypes.

An alternative approach to using the `%exception` directive is to use the "except" typemap. This allows you to attach an error handler to specific datatypes and function name. The typemap is applied to the return value of a function. For example :

```
%typemap(except) void *malloc {
    $action
    if (!$1) {
        PyExc_SetString(PyExc_MemoryError,"Out of memory in $symname");
        return NULL;
    }
}

void *malloc(int size);
```

When applied, this automatically checks the return value of `malloc()` and raises an exception if it's invalid. For example :

```
>>> from example import *
>>> a = malloc(2048)
>>> b = malloc(1500000000)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
MemoryError: Out of memory in malloc
```

7 Exceptions, Features, and other Customizations

>>>

When "except" typemaps are used, they override any exception handler defined with `%exception`.

Compatibility note: The except typemap dates to earlier SWIG releases 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. The except typemap is preserved for now, but may be deprecated in future versions.

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.

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 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.

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");
```

`%feature` is a relatively new addition to SWIG that was not added until version 1.3.10. Its intended use is as a highly flexible customization mechanism that can be used to annotate declarations with additional information for

use by specific target language modules. For example, in the Python module, you might use `%feature` to rewrite 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);
}
```

As of this writing, `%feature` is still experimental. Further details of its use will be described in the documentation for specific language modules.

SWIG 1.3 – Last Modified : December 9, 2001

8 SWIG and Perl5

Caution: This chapter is under repair! In this chapter, we discuss SWIG's support of Perl5. While 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.

Preliminaries

In order for this section to make much sense, you will need to have Perl5.002 (or later) installed on your system. You should also determine if your version of Perl has been configured to use dynamic loading or not. SWIG will work with or without it, but the compilation process will be different.

Running SWIG

To build a Perl5 module, run swig using the `-perl5` option as follows :

```
swig -perl5 example.i
```

This will produce 3 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 into Perl. The third file will be a documentation file (the exact filename depends on the documentation style). To finish building a module, you will need to compile the file `example_wrap.c` and link it with the rest of your program (and possibly Perl itself). There are several methods for doing this.

Getting the right header files

In order to compile, you will need to use the following Perl5 header files :

```
#include "Extern.h"
#include "perl.h"
#include "XSUB.h"
```

These are usually located in a directory like this

```
/usr/local/lib/perl5/arch-osname/5.003/CORE
```

The SWIG configuration script will try to find this directory, but it's not entirely foolproof. You may have to dig around yourself.

Compiling a dynamic module

To compile a dynamically loadable module, you will need to do something like the following,

```
% gcc example.c
% gcc example_wrap.c -I/usr/local/lib/perl5/arch-osname/5.003/CORE
    -Dbool=char -c
% ld -shared example.o example_wrap.o -o example.so    # Irix
```

8 SWIG and Perl5

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 name on your system (check the man pages for the linker).

Unfortunately, the process of building dynamic modules varies on every single machine. Both the C compiler and linker may need special command line options. SWIG tries to guess how to build dynamic modules on your machine in order to run its example programs. Again, the configure script isn't foolproof.

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 simple 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
```

This is the preferred approach if you building general purpose Perl5 modules for distribution. More information about MakeMaker can be found in "Programming Perl, 2nd ed." by Larry Wall, Tom Christiansen, and Randal Schwartz.

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 -perl5 -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 -perl5 -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 a 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 as follows :

```
% ld example.o example_wrap.o -L/usr/local/lib/perl5/arch/5.003/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...

Compilation problems and compiling with C++

In some cases, you may get alot of error messages about the `'bool'` datatype when compiling a SWIG 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.

Compiling dynamic modules for C++ is also a tricky business. When compiling C++ modules, you may need to link using the C++ compiler such as :

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

It may also be necessary to link against the `libgcc.a`, `libg++.a`, and `libstdc++.a` libraries (assuming g++). C++ may also complain about one line in the Perl header file "`perl.h`" and the invalid use of the "explicit" keyword. To work around this problem, put the option `-Dexplicit=` in your compiler flags.

If all else fails, put on your wizard cap and start looking around in the header files. Once you've figured out how to get one module to compile, you can compile just about all other modules.

Building Perl Extensions under Windows 95/NT

Building a SWIG extension to Perl 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 Perl interpreter. This section assumes you are using SWIG with Microsoft Visual C++ 4.x although the procedure may be similar with other compilers. SWIG currently supports the ActiveWare Perl for Win32 port and Perl 5.004. If using the ActiveWare version, all Perl extensions must be compiled using C++!

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 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 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";
```

It appears that DLL's will work if they are placed in the current working directory. To make a generally DLL available, it should be place (along with its support files) in the `Lib\Auto\[module]` sub-directory of the Perl directory where [module] is the name of your module.

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 an ActiveWare Perl for Win32 extension
# Note : Extensions must be compiled with the C++ compiler!

SRCS      = example.cxx
IFILE     = example
INTERFACE = $(IFILE).i
```

```

WRAPFILE      = $(IFILE)_wrap.cxx

# 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
winpool.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 /W3 /nologo

# Perl 5.004
PERL_INCLUDE = -Id:\perl5\lib\CORE
PERLLIB      = d:\perl5\lib\CORE\perl.lib
PERLFLAGS    = /DWIN32 /DMSWIN32 /DWIN32IO_IS_STDIO

# ActiveWare
PERL_INCLUDE = -Id:\perl -Id:\perl\inc
PERL_LIB     = d:\perl\Release\perl300.lib
PERLFLAGS    = /DWIN32 /DMSWIN32 /DPERL_OBJECT

perl::
    ..\..\swig -perl5 -o $(WRAPFILE) $(INTERFACE)
    $(CC) $(CFLAGS) $(PERLFLAGS) $(PERL_INCLUDE) $(SRCS) $(WRAPFILE)
    set LIB=$(TOOLS)\lib
    $(LINK) $(LOPT) -out:example.dll $(LIBS) $(PERLLIB) example.obj
    example_wrap.obj

```

To build the extension, run NMAKE (note that you may be to run vcvars32 before doing this to set the correct environment variables). This is a simplistic Makefile, but hopefully its enough to get you started.

Modules, packages, and classes

When you create a SWIG extension, everything gets thrown together into a single Perl5 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

```

8 SWIG and Perl5

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 a lot 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.

By default, SWIG installs its functions into a package with the same name as the module. This can be changed by giving SWIG the `-package` option :

```
% swig -perl5 -package FooBar example.i
```

In this case, we still create a module called ``example'`, but all of the functions in that module will be installed into the package ``FooBar.'` For example :

```
use example;                # Load the module like before
print FooBar::fact(4), "\n"; # Call a function in package FooBar
```

Perl supports object oriented programming using packages. A package can be thought of as a namespace for a class containing methods and data. The reader is well advised to consult "Programming Perl, 2nd Ed." by Wall, Christiansen, and Schwartz for most of the gory details.

Basic Perl interface

Functions

C functions are converted into new Perl commands (or subroutines). Default/optional arguments are also allowed. An interface file like this :

```
%module example
int foo(int a);
double bar (double, double b = 3.0);
...
```

Will be used in Perl like this :

```
use example;
$a = &example::foo(2);
$b = &example::bar(3.5, -1.5);
$c = &example::bar(3.5);          # Use default argument for 2nd parameter
```

Okay, this is pretty straightforward...enough said.

Global variables

Global variables are handled using pure magic—well, Perl's magic variable mechanism that is. In a nutshell, it is possible to make certain Perl variables "magical" by attaching methods for getting and setting values among other things. SWIG generates a pair of functions for accessing C global variables and attaches them to a Perl variable of the same name. 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 ...
```

SWIG supports global variables of all C datatypes including pointers and complex objects.

Constants

Constants are created as read-only magical variables and operate in exactly the same manner.

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);
```

SWIG will return a value as if you had done this :

```
$ptr = new_Matrix(int n, int m);    # Save pointer return result
bless $ptr, "MatrixPtr";           # Bless it as a MatrixPtr
```

SWIG uses the "blessing" to check the datatype of various pointers. In the event of a mismatch, an error or warning message will be 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.";
}
```

```
}
```

It is easy to get burned by references in more subtle ways. For example, if you are storing a hash table of objects, it may be best to use the actual C pointer value rather than the Perl reference as a key. Since each Perl reference to the same C object may be different, it would be impossible to find anything in the hash without this. As a general rule, the best way to avoid problems with references is to make sure hash tables, comparisons, and other pointer operations use the value of the reference (ie. `$$a`), not the reference itself.

Structures and C++ classes

For structures and classes, SWIG produces a set of accessor functions for member functions and member data. For example :

```
%module vector

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

This gets turned into the following collection of Perl functions :

```
vector::Vector_x_get($obj);
vector::Vector_x_set($obj,$x);
vector::Vector_y_get($obj);
vector::Vector_y_set($obj,$y);
vector::Vector_z_get($obj);
vector::Vector_z_set($obj,$z);
vector::new_Vector();
vector::delete_Vector($obj);
vector::Vector_magnitude($obj);
```

To use the class, simply use these functions. As it turns out, SWIG has a mechanism for creating shadow classes that hides these functions and uses an object oriented interface instead—keep reading.

Examples

Write me.

Exception handling

The SWIG `%except` 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 exception handling 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();
        }
    }
};

```

The functions associated with this class can throw a range exception for an out-of-bounds array access. We can catch this in our Perl extension by specifying the following in an interface file :

```

%except(perl5) {
    try {
        $function                // Gets substituted by actual function call
    }
    catch (RangeError) {
        croak("Array index out-of-bounds");
    }
}

```

Now, when the C++ class throws a RangeError exception, our wrapper functions will catch it, turn it into a Perl exception, and allow a graceful death as opposed to just having some sort of mysterious program crash. Since SWIG's exception handling is user-definable, we are not limited to C++ exception handling. It is also possible to write a language-independent exception handler that works with other scripting languages. Please see the chapter on exception handling for more details.

Remapping datatypes with typemaps

While SWIG does a reasonable job with most C/C++ datatypes, it doesn't always do what you want. However, you can remap SWIG's treatment of just about any datatype using a typemap. A typemap simply specifies a conversion between Perl and C datatypes and can be used to process function arguments, function return values, and more. A similar mechanism is used by the xsubpp compiler provided with Perl although the SWIG version provides many more options.

A simple typemap example

The following example shows how a simple typemap can be written :

```
%module example

%typemap(perl5,in) int {
    $target = (int) SvIV($source);
    printf("Received an integer : %d\n", $target);
}
...
extern int fact(int n);
```

Typemap specifications require a language name, method name, datatype, and conversion code. For Perl5, "perl5" should be used as the language name. The "in" method refers to the input arguments of a function. The 'int' datatype tells SWIG that we are remapping integers. The conversion code is used to convert from a Perl scalar value to the corresponding datatype. Within the support code, the variable `$source` contains the source data (a Perl object) and `$target` contains the destination of the conversion (a C local variable).

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
```

General discussion of typemaps can be found in the main SWIG users reference.

Perl5 typemaps

The following typemap methods are available to Perl5 modules

`%typemap(perl5,in)` Converts Perl5 object to input function arguments.

`%typemap(perl5,out)` Converts function return value to a Perl5 value.

`%typemap(perl5,varin)` Converts a Perl5 object to a global variable.

`%typemap(perl5,varout)` Converts a global variable to a Perl5 object.

`%typemap(perl5, freearg)` Cleans up a function argument after a function call

`%typemap(perl5, argout)` Output argument handling

`%typemap(perl5, 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(perl5, check)` Check value of input parameter.

Typemap variables

The following variables may be used within the C code used in a typemap :

`$source` Source value of a conversion

`$target` Target of conversion (where result should be stored)

`$type` C datatype being remapped

`$mangle` Mangled version of datatype (for blessing objects)

`$arg` Function argument (when applicable).

Name based type conversion

Typemaps are based both on the datatype and an optional name attached to a datatype. For example :

```
%module foo

// This typemap will be applied to all char ** function arguments
%typemap(perl5,in) char ** { ... }

// This typemap is applied only to char ** arguments named `argv'
%typemap(perl5,in) char **argv { ... }
```

In this example, two typemaps are applied to the `char **` datatype. However, the second typemap will only be applied to arguments named ``argv'`. A named typemap will always override an unnamed typemap.

Due to the name matching scheme, typemaps do not follow typedef declarations. For example :

```
%typemap(perl5,in) double {
    ... get a double ...
}

double foo(double);           // Uses the double typemap above
typedef double Real;
Real bar(Real);               // Does not use the typemap above (double != Real)
```

Is is odd behavior to be sure, but you can work around the problem using the %apply directive as follows :

```
%typemap(perl5,in) double {
    ... get a double ...
}
double foo(double);                // Uses the double typemap above

typedef double Real;
%apply double { Real };            // Apply the double typemap to Reals.
Real bar(Real);                    // Uses the double typemap already defined.
```

Named typemaps are extremely useful for managing special cases. It is also possible to use named typemaps to process output arguments (ie. function arguments that have values returned in them).

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

// This tells SWIG to treat char ** as a special case
%typemap(perl5,in) char ** {
    AV *tempav;
    I32 len;
    int i;
    SV **tv;
    if (!SvROK($source))
        croak("$source is not a reference.");
    if (SvTYPE(SvRV($source)) != SVt_PVAV)
        croak("$source is not an array.");
    tempav = (AV*)SvRV($source);
    len = av_len(tempav);
    $target = (char **) malloc((len+2)*sizeof(char *));
    for (i = 0; i <= len; i++) {
        tv = av_fetch(tempav, i, 0);
        $target[i] = (char *) SvPV(*tv,na);
    }
    $target[i] = 0;
};

// This cleans up our char ** array after the function call
%typemap(perl5,freearg) char ** {
    free($source);
}

// Creates a new Perl array and places a char ** into it
%typemap(perl5,out) char ** {
    AV *myav;
    SV **svs;
    int i = 0, len = 0;
    /* Figure out how many elements we have */
    while ($source[len])
        len++;
    svs = (SV **) malloc(len*sizeof(SV *));
    for (i = 0; i < len ; i++) {
        svs[i] = sv_newmortal();
```

```

        sv_setpv((SV*)svs[i],$source[i]);
    };
    myav = av_make(len,svs);
    free(svs);
    $target = newRV((SV*)myav);
    sv_2mortal($target);
}

// 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, our wrapped C functions can be used in a Perl script as follows :

```

use argv;
@a = ("Dave", "Mike", "John", "Mary");           # Create an array of strings
argv::print_args(\@a);                           # Pass it to our C function
$b = argv::get_args();                           # Get array of strings from C
print @$b,"\n";                                   # Print it out

```

Of course, there are many other possibilities. As an alternative to array references, we could pass in strings separated by some delimiter and perform a splitting operation in C.

Using typemaps to return values

Sometimes it is desirable for a function to return a value in one of its arguments. A named typemap can be used to handle this case. For example :

```

%module return

// This tells SWIG to treat an double * argument with name 'OutDouble' as
// an output value.

%typemap(perl5,argout) double *OutDouble {
    $target = sv_newmortal();
    sv_setnv($target, *$source);
    argvi++;                               /* Increment return count -- important! */
}

// If we don't care what the input value is, we can make the typemap ignore it.

%typemap(perl5,ignore) double *OutDouble(double junk) {
    $target = &junk;                       /* junk is a local variable that has been declared */
}

```

```

    }

    // 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 *OutDouble, double *OutDouble);
    ...

```

When output arguments are encountered, they are simply appended to the stack used to return results. This will show up as an array when used in Perl. For example :

```

    @r = multout(7,13);
    print "multout(7,13) = @r\n";

```

Accessing array structure members

Consider the following data structure :

```

#define NAMELEN    32
typedef struct {
    char    name[NAMELEN];
    ...
} Person;

```

By default, SWIG doesn't know how to handle the name structure since it's an array, not a pointer. In this case, SWIG will make the array member readonly. However, member typemaps can be used to make this member writable from Perl as follows :

```

%typemap(memberin) char[NAMELEN] {
    /* Copy at most NAMELEN characters into $target */
    strncpy($target,$source,NAMELEN);
}

```

Whenever a `char[NAMELEN]` type is encountered in a structure or class, this typemap provides a safe mechanism for setting its value. An alternative implementation might choose to print an error message if the name was too long to fit into the field.

It should be noted that the `[NAMELEN]` array size is attached to the typemap. A datatype involving some other kind of array would be affected. However, we can write a typemap that will work for any array dimension as follows :

```

%typemap(memberin) char [ANY] {
    strncpy($target,$source,$dim0);
}

```


When code is generated, `$dim0` gets filled in with the real array dimension.

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)
```

Unfortunately, this does NOT work. There are many reasons for this, but the main one is that SWIG has no idea what a `double *` really is. It could be an input value, an output value, or an array of 2 million elements. As a result, SWIG leaves it alone and looks exclusively for a C pointer value (which is not the same as a Perl reference—well, at least note of the type used in the above script).

However, you can use a `typemap` to get the desired effect. For example :

```
%typemap(perl5,in) double * (double dvalue) {
    SV* tempSV;
    if (!SvROK($source)) {
        croak("expected a reference\n");
    }
    tempSV = SvRV($source);
    if ((!SvNOK(tempSV)) && (!SvIOK(tempSV))) {
        croak("expected a double reference\n");
    }
    dvalue = SvNV(tempSV);
    $target = &dvalue;
}

%typemap(perl5,argout) double * {
    SV *tempSV;
    tempSV = SvRV($arg);
    sv_setnv(tempSV, *$source);
}
```

Now, if you place this before our `add` function, we can do this :

```
$a = 3.5;
$b = 7.5;
$c = 0.0;
add($a,$b,\$c);          # Now it works!
print "$c\n";
```

You'll get the output value of "11.0" which is exactly what we wanted. While this is pretty cool, it should be stressed that you can easily shoot yourself in the foot with typemaps—of course SWIG is has never been too concerned about legislating morality....

Useful functions

When writing typemaps, it is necessary to work directly with Perl5 objects. This, unfortunately, can be a daunting task. Consult the "perl5guts" 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 usef quite effectively without knowing any of these details—especially now that there are typemap libraries that can already been 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 *, int len);
void     sv_setpv(SV *, char *val);
void     sv_setpv(SV *, char *val, int len);
SV       *newSVpv(char *value, int len);
int      SvPOK(SV *);
void     sv_catpv(SV *, char *);
void     sv_catpv(SV *, char *, int);
```

Perl References

```
void     sv_setref_pv(SV *, char *, void *ptr);
int      sv_isobject(SV *);
SV       *SvRV(SV *);
int      sv_isa(SV *, char *0);
```

Standard typemaps

The following typemaps show how to convert a few common types of objects between Perl and C (and to give a better idea of how everything works).

(rewrite)

Pointer handling

SWIG pointers are represented as blessed references. The following functions can be used to create and read pointer values.

(rewrite)

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(perl5,argout) int *OUTPUT {
    if (argvi >= items) {
        EXTEND(sp,1);          /* Extend the stack by 1 object */
    }
    $target = sv_newmortal();
    sv_setiv($target,(IV) *($source));
    argvi++;
}
```

The gory details on shadow classes

Perl5 shadow classes are constructed on top of the low-level C interface provided by SWIG. By implementing the classes in Perl instead of C, we get a number of advantages :

- The classes are easier to implement (after all Perl makes lots of things easier).
- By writing in Perl, the classes tend to interact better with other Perl objects and programs.
- You can modify the resulting Perl code without recompiling the C module.

Shadow classes are new in SWIG 1.1 and still somewhat experimental. The current implementation is a combination of contributions provided by Gary Holt and David Fletcher—many thanks!

Module and package names

When shadow classing is enabled, SWIG generates a low-level package named ``modulec'` where ``module'` is the name of the module you provided with the `%module` directive (SWIG appends a ``c'` to the name to indicate that it is the low-level C bindings). This low-level package is exactly the same module that SWIG would have created without the ``-shadow'` option, only renamed.

Using the low-level interface, SWIG creates Perl wrappers around classes, structs, and functions. This collection of wrappers becomes the Perl module that you will use in your Perl code, not the low-level package (the original package is hidden, but working behind the scenes).

What gets created?

Suppose you have the following SWIG interface file :

```
%module vector
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.

```
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);
```

These functions can now be used to create a Perl shadow class that looks like this :

```
package Vector;
@ISA = qw( vector );
%OWNER = ();
%BLESSEDMEMBERS = ();

sub new () {
    my $self = shift;
    my @args = @_;
    $self = vectorc::new_Vector(@args);
    return undef if (!defined($self));
    bless $self, "Vector";
    $OWNER{$self} = 1;
    my %retval;
    tie %retval, "Vector", $self;
    return bless \%retval, "Vector";
}

sub DESTROY {
    my $self = shift;
    if (exists $OWNER{$self}) {
        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 shadow 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();

```

Object Ownership

In order for shadow 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.

8 SWIG and Perl5

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 shadow 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.

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 shadow 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);
```

Shadow Functions

When functions take arguments involving a complex object, it is sometimes necessary to write a shadow function. For example :

```
double dot_product(Vector *v1, Vector *v2);
```

Since Vector is an object already wrapped into a shadow 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.

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 {
```

8 SWIG and Perl5

```
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 shadow classes are implemented only once and reused in the wrapper code for various classes and structures.

Since SWIG shadow 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).

Iterators

With each class or structure, SWIG also generates a pair of functions to support Perl iterators. This makes it possible to use the `keys` and `each` functions on a C/C++ object. Iterators are implemented using code like this :

```
sub FIRSTKEY {
    my $self = shift;
    @ITERATORS{$self} = ['x','y','z', ];
    my $first = shift @{$ITERATORS{$self}};
    return $first;
}

sub NEXTKEY {
    my $self = shift;
    $nelem = scalar @{$ITERATORS{$self}};
    if ($nelem > 0) {
        my $member = shift @{$ITERATORS{$self}};
        return $member;
    } else {
        @ITERATORS{$self} = ['x','y','z', ];
        return ();
    }
}
```

The %ITERATORS hash table maintains the state of each object for which the `keys` or `each` function has been

applied to. The state is maintained by keeping a list of the member names.

While iterators may be of limited use when working with C/C++ code, it turns out they can be used to perform an element by element copy of an object.

```
$v = new Vector(1,2,3);  
$w = new Vector(0,0,0);  
%$w = %$v;           # Copy contents of v into w
```

However, this is not a deep copy so they probably works better with C than with C++.

Where to go from here?

The SWIG Perl5 module is constantly improving to provide better integration with Perl and to be easier to use. The introduction of shadow classes and typemaps in this release are one more step in that direction. The SWIG Examples directory contains more simple examples of building Perl5 modules. As always, suggestions for improving the Perl5 implementation are welcome.

SWIG 1.1 – Last Modified : Mon Aug 4 10:47:01 1997

9 SWIG and Python

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. The original release of SWIG was developed for Python 1.3 so current versions may still work with that release. However, this hasn't been tested for quite some time and your mileage might vary. For the best results, consider using Python 2.0 or newer.

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.

Preliminaries

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 a file `example_wrap.c` or `example_wrap.cxx` that contains all of the code needed to build a Python extension module. To finish building the module, you need to compile this file and link it with the rest of your program.

Getting the right header files

In order to compile the wrapper code, 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
>>>
```

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 -python example.i
```

9 SWIG and Python

```
$ gcc -c example.c
$ gcc -c example_wrap.c -I/usr/local/include/python2.0
$ gcc -shared example.o example_wrap.o -o examplemodule.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/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. If the name of your SWIG module is "example", the name of the corresponding object file should be "examplemodule.so" or "example.so". The name of the module is specified using the `%module` directive or the `-module` command line option.

Using distutils

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).

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 "", line 1, in ?
ImportError: dynamic module does not define init function (initexample)
>>>
```

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:

```
Traceback (most recent call last):
  File "example.py", line 3, in ?
    import example
ImportError: ./examplemodule.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:

9 SWIG and Python

```
$ gcc -shared example.o example_wrap.o -L/home/beazley/projects/lib -lfoo \
-o examplemodule.so
```

If the `foo` library is compiled as a shared library, you might get the following problem when you try to use your module:

```
>>> import example
Traceback (most recent call last):
  File "", 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 examplemodule.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` to add additional search paths to the default system configuration (this requires root access and you will need to read the man pages).

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 examplemodule.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 -L/opt/SUNWspro/lib -o examplemodule.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.

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).

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](#).

The primitive Python–C interface

At its core, the Python module uses a simple low–level interface to C function, variables, constants, and classes. This low–level interface is then used to create more user–friendly interfaces such as shadow classes. This section describes the low–level interface.

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. When choosing a module name, make sure you don't use the same name as a built–in Python command or standard module name. Otherwise, the results may be unpredictable.

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 like this:

```
>>> import example
>>> print example.fact(4)
24
>>>
```

Variable Linking

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
```

9 SWIG and Python

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 "", 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 also use the `%readonly` directive. For example:

```
%readonly
extern char *path;
%readwrite
```

The `%readonly` directive stays in effect until it is explicitly disabled using `%readwrite`.

If you would like to use a name other than "cvar", it can be changed using the `-globals` option :

```
% swig -python -globals myvar example.i
```

Finally, 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. Also, SWIG does not create `cvar` if there are no global variables in a module.

Constants

C/C++ constants are installed as Python objects containing 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";
```

Note: C 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.

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.

Pointers

Pointers to C/C++ objects are represented as encoded character strings such as the following :

```
_800f8e28_p_Vector
```

A NULL pointer is represented by the Python None object.

As an alternative to strings, SWIG can also 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 examplemodule.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
>>> a = example.new_Circle(10)
>>> b = example.new_Square(20)
>>> a
<PyCObject object at 0x80c5e60>
>>> b
<PyCObject object at 0x8107800>
>>>
```

Notice how no clues regarding the type of `a` and `b` is shown. On the other hand, the string representation produces the following:

```
>>> a
'_88671008_p_Circle'
>>> b
'_605f0c08_p_Square'
>>>
```

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;
}
```

9 SWIG and Python

```
/* C++-style cast */
Foo *BarToFoo(Bar *b) {
    return dynamic_cast<Foo*>(b);
}

Foo *IncrFoo(Foo *f, int i) {
    return f+i;
}
%}
```

If you need to type-cast a lot of objects, it may indicate a serious weakness in your design. 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.

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 Python as follows:

```
>>> v = new_Vector()
>>> Vector_x_get(v)
3.5
>>> Vector_x_set(v,7.8)          # Change x component
>>> print Vector_x_get(v), Vector_y_get(v), Vector_z_get(v)
7.8 -4.5 0.0
>>>
```

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 %readonly directive. For example:

```
struct Foo {
    ...
    %readonly
    int x;          /* Read-only members */
    char *name;
```

```

        %readwrite
        ...
    };

```

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,

```

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
}

void Bar_f_set(Bar *b, Foo *val) {
    b->f = *val;
}

```

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);
};

```

9 SWIG and Python

```
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 Python, these functions are used as follows:

```
>>> l = new_List()
>>> List_insert(l,"Ale")
>>> List_insert(l,"Stout")
>>> List_insert(l,"Lager")
>>> List_print(l)
Lager
Stout
Ale
>>> print List_length_get(l)
3
>>> print l
_80085608_p_List
>>>
```

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 known as shadow classes can be built using these low-level accessors. This is described shortly.

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).

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 Python, the methods are accessed as follows:

```
>>> import example
>>> example.foo_i(3)
>>> s = example.new_Spam()
>>> Spam.foo_i(s,3)
>>> Spam.foo_d(s,3.14)
```

Please refer to the "SWIG Basics" chapter for more information.

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 Python identifier. For example:

```
%rename(add_complex) operator+(Complex ,Complex )
...
Complex operator+(Complex ,Complex )
```

Now, in Python, you can do this:

```
>>> a = example.new_Complex(2,3)
>>> b = example.new_Complex(4,-1)
>>> c = example.add_complex(a,b)
```

More details about wrapping C++ operators into Python operators is discussed a little later.

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) {
```

9 SWIG and Python

```
    *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)
```

Simple exception handling

The SWIG `%exception` directive can be used to define a user-definable exception handler that can convert C/C++ errors into Python exceptions. The chapter on exception handling contains more details, but suppose you have a C++ class like the following :

```
class RangeError {};    // Used for an exception

class DoubleArray {
private:
```

9 SWIG and Python

```
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 Python extension by writing the following in an interface file:

```
%exception {
    try {
        $action
    }
    catch (RangeError) {
        PyErr_SetString(PyExc_IndexError, "index out-of-bounds");
        return NULL;
    }
}

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 raise a Python 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. 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) {
        PyErr_SetString(PyExc_IndexError,"index out-of-bounds");
        return NULL;
    }
}

%exception setitem {
    try {
        $action
    }
    catch (RangeError) {
        PyErr_SetString(PyExc_IndexError,"index out-of-bounds");
        return NULL;
    }
}

```

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) {
        PyErr_SetString(PyExc_IndexError,"index out-of-bounds");
        return NULL;
    }
}
%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 ["Exception Handling"](#) for more examples.

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

```

9 SWIG and Python

```
PyExc_NameError
PyExc_NotImplementedError
PyExc_OSError
PyExc_OverflowError
PyExc_RuntimeError
PyExc_StandardError
PyExc_SyntaxError
PyExc_SystemError
PyExc_TypeError
PyExc_UnicodeError
PyExc_ValueError
PyExc_ZeroDivisionError
```

These exceptions are actually organized into an hierarchy as shown below. Consult the Python Essential Reference for more details (shameless plug):

```
PyExc_Exception
  PyExc_SystemExit
  PyExc_StandardError
    PyExc_ArithmeticError
      PyExc_FloatingPointError
      PyExc_OverflowError
      PyExc_ZeroDivisionError
    PyExc_AssertionError
    PyExc_AttributeError
    PyExc_EnvironmentError
      PyExc_IOError
      PyExc_OSError
    PyExc_EOFError
    PyExc_ImportError
    PyExc_KeyboardInterrupt
    PyExc_LookupError
      PyExc_IndexError
      PyExc_KeyError
    PyExc_MemoryError
    PyExc_NameError
    PyExc_RuntimeError
      PyExc_NotImplementedError
    PyExc_SyntaxError
    PyExc_SystemError
    PyExc_TypeError
    PyExc_ValueError
      PyExc_UnicodeError
```

Compatibility note: In SWIG1.1, exceptions were defined using the older `%except` directive:

```
%except(python) {
  try {
    $action
  }
  catch (RangeError) {
    PyErr_SetString(PyExc_IndexError,"index out-of-bounds");
    return NULL;
  }
}
```

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.

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.

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 n {
    $1 = (int) PyLong_AsLong($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:

9 SWIG and Python

```
%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
```

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) {
    $1 = PyString_AsString($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
>>>
```

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);
}
```

The following list details all of the typemap methods that can be used by the Python module:

`%typemap(in)`

Converts Python objects to input function arguments

`%typemap(out)`

Converts return value of a C function to a Python object

`%typemap(varin)`

Assigns a C global variable from a Python object

`%typemap(varout)`

Returns a C global variable as a Python 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 Python constants (constant table)

`%typemap(constcode)`

Creation of Python 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(ignore)`

Ignore an argument, but initialize its value.

`%typemap(arginit)`

Initialize an argument to a value before any conversions occur.

Examples of these methods will appear shortly.

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:

9 SWIG and Python

\$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.

Useful 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

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.

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.

9 SWIG and Python

```
%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.

Expanding a Python 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 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"])
```

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 */
```

9 SWIG and Python

```
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(ignore) double *OutValue(double temp) {
    $1 = &temp;
}
```

The ignore typemap forces the input value to be 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)
>>>
```

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.

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");
        }
    }
}
```

9 SWIG and Python

```
        return NULL;
    }
    temp[i] = PyFloat_AsDouble(o);
}
$1 =
}
```

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 =
}
```

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.

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. If set, the function converts type-errors into a Python `TypeError` exception. If set to zero, no Python exception is raised. 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

9 SWIG and Python

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 **) SWIGTYPE_p_Foo, 1) == -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_descriptor, 1)) == -1) return NULL;
}
```

Although the pointer handling functions are primarily intended for manipulating low-level pointers, both functions are fully aware of Python shadow classes (described shortly). Specifically, `SWIG_ConvertPtr()` will retrieve a pointer from any object that has a `this` attribute. In addition, `SWIG_NewPointerObj()` can automatically generate a shadow class object (if applicable).

Other odds and ends

Adding native Python functions to a SWIG module

Sometimes it is desirable to add a native Python method to a SWIG wrapper file. Suppose you have the following Python/C function :

```
PyObject *spam_system(PyObject *self, PyObject *args) {
    char *command;
    int sts;
    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

This function can be added to a SWIG module using the following declaration :

```
%native(system) spam_system;           // Create a command called `system`
```

Alternatively, you can use the full function declaration like this

```
%native(system) PyObject *spam_system(PyObject *self, PyObject *args);
```

or

```
%native(system) extern PyObject *spam_system(PyObject *self, PyObject *args);
```


Python shadow classes

Using the low-level C interface, it is possible to create a Python class that works like the original C++ class. In this case, the Python class is said to shadow the original C++ class. It's really just a wrapper around the C++ original class.

A simple example

One way to illustrate a shadow class is to write one by hand. For example:

```
class List:
    def __init__(self):
        self.this = new_List()
    def __del__(self):
        delete_List(self.this)
    def search(self,item):
        return List_search(self.this,item)
    def insert(self,item):
        List_insert(self.this,item)
    def remove(self,item):
        List_remove(self.this,item)
    def get(self,n):
        return List_get(self.this,n)
    def __getattr__(self,name):
        if name == "length" : return List_length_get(self.this)
        else : return self.__dict__[name]
    def __setattr__(self,name,value):
        if name == "length": List_length_set(self.this,value)
        else : self.__dict__[name] = value
```

In the shadow class, a reference to the underlying C++ object is kept in the `.this` attribute. Methods are then written so that they pass the `.this` attribute along with the other arguments to the low-level accessor function created by SWIG. This, in turn, allows the class to be used like this:

```
>>> l = List()
>>> l.insert("Ale")
>>> l.insert("Stout")
>>> l.insert("Lager")
>>> List_print(l.this)
Lager
Stout
Ale
>>> l.length
3
```

Clearly, this is a much nicer interface than before—and it only required a small amount of Python coding.

Why shadow classes?

Shadow classes are not the only approach to wrapping C++ classes. An alternative approach would be to wrap classes as new Python types in C. However, this requires a substantial amount of complicated C wrapper code. Furthermore, this approach makes it difficult to correctly handle inheritance and other advanced language features. Another more serious problem is that until recently, Python types created in C could not be subclassed or used in the same way as you would use a real Python class. As a result, C++ classes wrapped in this manner are

9 SWIG and Python

crippled versions of what you would obtain by writing a real class in Python.

Shadow classes allow C++ objects to be easily wrapped by a real Python class. This means that all of the normal features such as inheritance work like you would expect. The fact that such classes are written in Python also simplifies coding since it is much easier to write a class interface in Python than it is in C. Such classes are also much easier to modify since changes can be made without having to recompile any of the low-level C extension code. The main downside to this approach is worse performance—a concern for some users.

Automatic shadow class generation

SWIG can automatically generate shadow classes if you use the `-shadow` option :

```
swig -python -shadow interface.i
```

This generates the usual wrapper file along with an extra file `module.py` that contains the Python shadow-class code. The name of this file is the same as specified by the `%module` directive.

Since the shadow class code needs to be placed in a different module than the low-level C wrappers, the primitive interface is placed into a Python extension module named `modulec` (a 'c' is appended to the module name). When a user imports the module, the `module.py` is loaded. This file, in turn, imports the low-level C wrapper module to gain access to the accessor functions. For example, in the list example, the shadow file might look roughly like this:

```
# example
import examplec

class List:
    def __init__(self):
        self.this = examplec.new_List()
    def __del__(self):
        examplec.delete_List(self.this)
    def search(self,item):
        return examplec.List_search(self.this,item)
    def insert(self,item):
        examplec.List_insert(self.this,item)
    def remove(self,item):
        examplec.List_remove(self.this,item)
    def get(self,n):
        return examplec.List_get(self.this,n)
    def __getattr__(self,name):
        if name == "length" : return examplec.List_length_get(self.this)
        else : return self.__dict__[name]
    def __setattr__(self,name,value):
        if name == "length": examplec.List_length_set(self.this,value)
        else : self.__dict__[name] = value
```

The choice of appending a 'c' to the module name is somewhat non-standard and may cause a module name conflict in certain cases. To fix this, you can run SWIG with the `-interface` option to change the name of the C module file. For example, this places the low-level C interface into a module named `_example`:

```
$ swig -python -shadow -c++ -interface _example example.i
```

When shadow classes are used, most users don't notice the existence of the low-level C accessors. In fact, there is very little reason to use the low-level functions directly as shadow classes provide all of the needed access.

Compiling modules with shadow classes

To compile a module involving shadow classes, you can use the same procedure as before except that the module name now has an extra `c' appended to the name. Thus, an interface file like this

```
%module example
... a bunch of declarations ...
```

might be compiled as follows :

```
% swig -python -shadow example.i
% gcc -c example.c example_wrap.c -I/usr/local/include/python1.4 \
    -I/usr/local/lib/python1.4/config -DHAVE_CONFIG_H
% ld -shared example.o example_wrap.o -o examplecmodule.so
```

Notice the naming of `examplecmodule.so' as opposed to `examplemodule.so' that would have been created without shadow classes.

Shadow classes and type-checking

Because shadow classes are used so frequently, the SWIG type-checker is programmed to accept a raw-pointer or any object that contains a `this` attribute (which is assumed to be a raw-pointer). Thus, the earlier example can be simplified as follows:

```
class List:
    def __init__(self):
        self.this = new_List()
    def __del__(self):
        delete_List(self)
    def search(self,item):
        return List_search(self,item)
    def insert(self,item):
        List_insert(self,item)
    def remove(self,item):
        List_remove(self,item)
    def get(self,n):
        return List_get(self,n)
    def __getattr__(self,name):
        if name == "length" : return List_length_get(self)
        else : return self.__dict__[name]
    def __setattr__(self,name,value):
        if name == "length": List_length_set(self,value)
        else : self.__dict__[name] = value
```

You can also observe this behavior of the type checker with a little experimentation. For example:

```
>>> import example
>>> a = example.new_List()
>>> a
'_90651008_p_List'
>>> example.List_insert(a,"Ale")
>>> class Blah: pass
...
>>> b = Blah()
```

9 SWIG and Python

```
>>> b.this = a
>>> example.List_insert(b, "Lager")
>>>
```

Further details of type-checking and shadow classes will be described later. Stay tuned.

A preview

Shadow classes are one of SWIG's most powerful features. However, to better understand some of the finer points, some other SWIG features need to first be described. Therefore, we return to the topic of shadow classes a little later in this chapter.

The gory details of shadow classes

This section describes the process by which SWIG creates shadow classes and some of the more subtle aspects of using them.

A simple shadow class

Consider the following declaration from our previous example :

```
%module pde
struct Grid2d {
    Grid2d(int ni, int nj);
    ~Grid2d();
    double **data;
    int      xpoints;
    int      ypoints;
};
```

The SWIG generated class for this structure looks like the following:

```
# This file was created automatically by SWIG.
import pdec
class Grid2dPtr :
    def __init__(self, this):
        self.this = this
        self.thisown = 0
    def __del__(self):
        if self.thisown == 1 :
            pdec.delete_Grid2d(self.this)
    def __setattr__(self, name, value):
        if name == "data" :
            pdec.Grid2d_data_set(self.this, value)
            return
        if name == "xpoints" :
            pdec.Grid2d_xpoints_set(self.this, value)
            return
        if name == "ypoints" :
            pdec.Grid2d_ypoints_set(self.this, value)
            return
        self.__dict__[name] = value
    def __getattr__(self, name):
        if name == "data" :
            return pdec.Grid2d_data_get(self.this)
        if name == "xpoints" :
```

```

        return pdec.Grid2d_xpoints_get(self.this)
    if name == "ypoints" :
        return pdec.Grid2d_ypoints_get(self.this)
    return self.__dict__[name]
def __repr__(self):
    return "<C Grid2d instance>"
class Grid2d(Grid2dPtr):
    def __init__(self, arg0, arg1) :
        self.this = pdec.new_Grid2d(arg0, arg1)
        self.thisown = 1

```

Module names

Shadow classes are built using the low-level SWIG generated C interface. This interface is named "modulec" where "module" is the name of the module specified in a SWIG interface file. The Python code for the shadow classes is created in a file "module.py". This is the file that should be loaded when a user wants to use the module.

Two classes

For each structure or class found in an interface file, SWIG creates two Python classes. If a class is named "Grid2d", one of these classes will be named "Grid2dPtr" and the other named "Grid2d". The Grid2dPtr class is used to turn wrap a Python class around an already preexisting Grid2d pointer. For example :

```

>>> gpPtr = create_grid2d()           # Returns a Grid2d from somewhere
>>> g = Grid2dPtr(gpPtr)              # Turn it into a Python class
>>> g.xpoints
50
>>>

```

The Grid2d class, on the other hand, is used when you want to create a new Grid2d object from Python. In reality, it inherits all of the attributes of a Grid2dPtr, except that its constructor calls the corresponding C++ constructor to create a new object. Thus, in Python, this would look something like the following :

```

>>> g = Grid2d(50,50)                # Create a new Grid2d
>>> g.xpoints
50
>>>

```

This two class model is a tradeoff. In order to support C/C++ properly, it is necessary to be able to create Python objects from both pre-existing C++ objects and to create entirely new C++ objects in Python. While this might be accomplished using a single class, it would complicate the handling of constructors considerably. The two class model, on the other hand, works, is consistent, and is relatively easy to use. In practice, you probably won't even be aware that there are two classes working behind the scenes.

The this pointer

Within each shadow class, the member "this" contains the actual C/C++ pointer to the object. You can check this out yourself by typing something like this :

```

>>> g = Grid2d(50,50)

```

9 SWIG and Python

```
>>> print g.this
_1008fe8_Grid2d_p
>>>
```

Direct manipulation of the "this" pointer is generally discouraged. In fact forget that you read this.

Object ownership

Ownership is a critical issue when mixing C++ and Python. For example, suppose I create a new object in C++, but later use it to create a Python object. If that object is being used elsewhere in the C++ code, we clearly don't want Python to delete the C++ object when the Python object is deleted. Similarly, what if I create a new object in Python, but C++ saves a pointer to it and starts using it repeatedly. Clearly, we need some notion of who owns what. Since sorting out all of the possibilities is probably impossible, SWIG shadow classes always have an attribute "thisown" that indicates whether or not Python owns an object. Whenever an object is created in Python, Python will be given ownership by setting `thisown` to 1. When a Python class is created from a pre-existing C/C++ pointer, ownership is assumed to belong to the C/C++ code and `thisown` will be set to 0.

Ownership of an object can be changed as necessary by changing the value of `thisown`. When set, Python will call the C/C++ destructor when the object is deleted. If it is zero, Python will never call the C/C++ destructor.

Constructors and Destructors

C++ constructors and destructors will be mapped into Python's `__init__` and `__del__` methods respectively. Shadow classes always contain these methods even if no constructors or destructors were available in the SWIG interface file. The Python destructor will only call a C/C++ destructor if `self.thisown` is set.

Member data

Member data of an object is accessed through Python's `__getattr__` and `__setattr__` methods.

Printing

SWIG automatically creates a Python `__repr__` method for each class. This forces the class to be relatively well-behaved when printing or being used interactively in the Python interpreter.

Shadow Functions

Suppose you have the following declarations in an interface file :

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

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

By default, the function `addv` will operate on `Vector` pointers, not Python classes. However, the SWIG Python module is smart enough to know that `Vector` has been wrapped into a Python class so it will create the following replacement for the `addv()` function.

```
def addv(a,b):
    result = VectorPtr(vectorc.addv(a.this,b.this))
    result.thisown = 1
    return result
```

Function arguments are modified to use the "this" pointer of a Python Vector object. The result is a pointer to the result which has been allocated by malloc or new (this behavior is described in the chapter on SWIG basics), so we simply create a new VectorPtr with the return value. Since the result involved an implicit malloc, we set the ownership to 1 indicating that the result is to be owned by Python and that it should be deleted when the Python object is deleted. As a result, operations like this are perfectly legal and result in no memory leaks :

```
>>> v = add(add(add(add(a,b),c),d),e)
```

Substitution of complex datatypes occurs for all functions and member functions involving structure or class definitions. It is rarely necessary to use the low-level C interface when working with shadow classes.

Nested objects

SWIG shadow classes support nesting of complex objects. For example, suppose you had the following interface file :

```
%module particle

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

typedef struct {
    Particle();
    ~Particle();
    Vector r;
    Vector v;
    Vector f;
    int    type;
} Particle;
```

In this case you will be able to access members as follows :

```
>>> p = Particle()
>>> p.r.x = 0.0
>>> p.r.y = -1.5
>>> p.r.z = 2.0
>>> p.v = addv(v1,v2)
>>> ...
```

Nesting of objects is implemented using Python's `__setattr__` and `__getattr__` functions. In this case, they would look like this :

```
class ParticlePtr:
    ...
    def __getattr__(self,name):
```

9 SWIG and Python

```
        if name == "r":
            return particlec.VectorPtr(Particle_r_get(self.this))
        elif name == "v":
            return particlec.VectorPtr(Particle_v_get(self.this))
        ...

    def __setattr__(self,name,value):
        if name == "r":
            particlec.Particle_r_set(self.this,value.this)
        elif name == "v":
            particlec.Particle_v_set(self.this,value.this)
        ...
```

The attributes of any given object are only converted into a Python object when referenced. This approach is more memory efficient, faster if you have a large collection of objects that aren't examined very often, and works with recursive structure definitions such as :

```
struct Node {
    char *name;
    struct Node *next;
};
```

Nested structures such as the following are also supported by SWIG. These types of structures tend to arise frequently in database and information processing applications.

```
typedef struct {
    unsigned int dataType;
    union {
        int      intval;
        double   doubleval;
        char     *charval;
        void     *ptrvalue;
        long     longval;
        struct {
            int    i;
            double f;
            void   *v;
            char  name[32];
        } v;
    } u;
} ValueStruct;
```

Access is provided in an entirely natural manner,

```
>>> v = new_ValueStruct()          # Create a ValueStruct somehow
>>> v.dataType
1
>>> v.u.intval
45
>>> v.u.longval
45
>>> v.u.v.v = _0_void_p
>>>
```


To support the embedded structure definitions, SWIG has to extract the internal structure definitions and use them to create new Python classes. In this example, the following shadow classes are created :

```
# Class corresponding to union u member
class ValueStruct_u :
    ...
# Class corresponding to struct v member of union u
class ValueStruct_u_v :
    ...
```

The names of the new classes are formed by appending the member names of each embedded structure.

Inheritance and shadow classes

Since shadow classes are implemented in Python, you can use any of the automatically generated classes as a base class for more Python classes. However, you need to be extremely careful when using multiple inheritance. When multiple inheritance is used, at most ONE SWIG generated shadow class can be involved. If multiple SWIG generated classes are used in a multiple inheritance hierarchy, you will get name clashes on the `this` pointer, the `__getattr__` and `__setattr__` functions won't work properly and the whole thing will probably crash and burn. Perhaps it's best to think of multiple inheritance as a big hammer that can be used to solve a lot of problems, but it hurts quite a lot if you accidentally drop it on your foot....

Methods that return new objects

By default SWIG assumes that constructors are the only functions returning new objects to Python. However, you may have other functions that return new objects as well. For example :

```
Vector *cross_product(Vector *v1, Vector *v2) {
    Vector *result = new Vector();
    result = ... compute cross product ...
    return result;
}
```

When the value is returned to Python, we want Python to assume ownership. The brute force way to do this is to simply change the value of `thisown`. For example :

```
>>> v = cross_product(a,b)
>>> v.thisown = 1                                # Now Python owns it
```

Unfortunately, this is ugly and it doesn't work if we use the result as a temporary value :

```
w = vector_add(cross_product(a,b),c)              # Results in a memory leak
```

However, you can provide a hint to SWIG when working with such a function as shown :

```
// C Function returning a new object
%new Vector *cross_product(Vector *v1, Vector *v2);
```

The `%new` directive only provides a hint that the function is returning a new object. The Python module will assign proper ownership of the object when this is used.

Performance concerns and hints

Shadow classing is primarily intended to be a convenient way of accessing C/C++ objects from Python. However, if you're directly manipulating huge arrays of complex objects from Python, performance may suffer greatly. In these cases, you should consider implementing the functions in C or thinking of ways to optimize the problem.

There are a number of ways to optimize programs that use shadow classes. Consider the following two code fragments involving the `Particle` data structure in a previous example :

```
def force1(p1,p2):
    dx = p2.r.x - p1.r.x
    dy = p2.r.y - p1.r.y
    dz = p2.r.z - p1.r.z
    r2 = dx*dx + dy*dy + dz*dz
    f = 1.0/(r2*math.sqrt(r2))
    p1.f.x = p1.f.x + f*dx
    p2.f.x = p2.f.x - f*dx
    p1.f.y = p1.f.y + f*dy
    p2.f.y = p2.f.y - f*dy
    p1.f.z = p1.f.z + f*dz
    p2.f.z = p2.f.z - f*dz

def force2(p1,p2):
    r1 = p1.r
    r2 = p2.r
    dx = r2.x - r1.x
    dy = r2.y - r1.y
    dz = r2.z - r1.z
    r2 = dx*dx + dy*dy + dz*dz
    f = 1.0/(r2*math.sqrt(r2))
    f1 = p1.f
    f2 = p2.f
    f1.x = f1.x + f*dx
    f2.x = f2.x - f*dx
    f1.y = f1.y + f*dy
    f2.y = f2.y - f*dy
    f1.z = f1.z + f*dz
    f2.z = f2.z - f*dz
```

The first calculation simply works with each `Particle` structure directly. Unfortunately, it performs a lot of dereferencing of objects. If the calculation is restructured to use temporary variables as shown in `force2`, it will run significantly faster—in fact, on my machine, the second code fragment runs more than twice as fast as the first one.

If performance is even more critical you can use the low-level C interface which eliminates all of the overhead of going through Python's class mechanism (at the expense of coding simplicity). When Python shadow classes are used, the low level C interface can still be used by importing the ``modulec'` module where ``module'` is the name of the module you used in the SWIG interface file.

10 SWIG and Tcl

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.

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.

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.

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.

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).

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).

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.

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).

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".

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`.

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 `tclsh` or `wish`. This section covers the process of using SWIG with Microsoft Visual C++, although the procedure may be similar with other compilers.

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
%
```

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

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
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.

Primitive Tcl Interface

This section describes the basic Tcl interface. The object-based interface is described shortly.

Functions

C functions are turned into new Tcl commands with the same usage as the C function. Default/optional arguments are also allowed. An interface file like this :

```

%module example
int foo(int a);
double bar (double, double b = 3.0);
...

```

Will be used in Tcl like this :

```

set a [foo 2]
set b [bar 3.5 -1.5]
set b [bar 3.5]           # Note : default argument is used

```

There isn't much more to say...this is pretty straightforward.

Global variables

For global variables, SWIG uses Tcl's variable tracing mechanism to provide direct access. For example:

10 SWIG and Tcl

```
// example.i
%module example
...
double My_variable;
...

# Tcl script
puts $My_variable          # Output value of C global variable
set My_variable 5.5        # Change the value
```

Compatibility Note: Variable tracing is currently supported for all C/C++ datatypes. In older versions of SWIG, only variables of type `int`, `double`, and `char *` could be linked. All other types were accessed using special function calls.

Constants

Constants are installed as new Tcl variables. For example:

```
%module example
#define FOO 42
```

is accessed as follows:

```
% puts $FOO
42
%
```

No attempt is made to enforce the read-only nature of a constant. Therefore, a user could reassign the value if they wanted. 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.

Pointers

Pointers to C/C++ objects are represented as character strings such as the following :

```
_100f8e2_p_Vector
```

A NULL pointer is represented by the string "NULL". NULL pointers can also be explicitly created as follows :

```
_0_p_Vector
```

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;
}
%}
```

If you need to type-cast a lot of objects, it may indicate a serious weakness in your design. 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.

Structures

SWIG generates a basic low-level interface to C structures. For example :

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

gets mapped into the following collection of C functions :

```
double Vector_x_get(Vector *obj)
```

10 SWIG and Tcl

```
double Vector_x_set(Vector *obj, double x)
double Vector_y_get(Vector *obj)
double Vector_y_set(Vector *obj, double y)
double Vector_z_get(Vector *obj)
double Vector_z_set(Vector *obj, double z)
```

These functions are then used in the resulting Tcl interface. For example :

```
# v is a Vector that got created somehow
% Vector_x_get $v
3.5
% Vector_x_set $v 7.8           # Change x component
```

Similar access is provided for unions and the data members of C++ classes.

C++ Classes

C++ classes are handled 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);
int        List_length_set(List *l, int n);
void       List_print(List *l);
```

Within Tcl, we can use the functions as follows :

```
% set l [new_List]
% List_insert $l Ale
% List_insert $l Stout
% List_insert $l Lager
% List_print $l
Lager
Stout
Ale
% puts [List_length_get $l]
3
% puts $l
```

```
_1008560_p_List
%
```

C++ objects are really just pointers (which are represented as strings). Member functions and data are accessed by simply passing a pointer into a collection of accessor functions that take the pointer as the first argument.

While somewhat primitive, the low-level SWIG interface provides direct and flexible access to almost any C++ object. As it turns out, it is possible to do some rather amazing things with this interface as will be shown in some of the later examples. SWIG also generates an object-based interface that can be used in addition to the basic interface just described here.

The object-based interface

In addition to the low-level accessors, SWIG also generates an object-based interface to C structures and C++ classes. This interface supplements the low-level SWIG interface already defined—in fact, both can be used simultaneously. To illustrate this interface, consider the previous `List` 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);
};
```

Using the object oriented interface requires no additional modifications or recompilation of the SWIG module (the functions are just used differently).

Creating new objects

The name of the class becomes a new command for creating an object. There are 5 methods for creating an object (`MyObject` is the name of the corresponding C++ class)

```
MyObject o                # Creates a new object named `o'

MyObject o -this $objptr  # Turn a pointer to an existing C++ object into a
                          # Tcl object named `o'

MyObject -this $objptr    # Turn the pointer $objptr into a Tcl "object"

MyObject -args args       # Create a new object and pick a name for it. A handle
                          # will be returned and is the same as the pointer value.

MyObject                  # The same as MyObject -args, but for constructors that
                          # take no arguments.
```

Thus, for our `List` class, you can create new `List` objects as follows :

```
List l                    # Create a new list l
```

10 SWIG and Tcl

```
set listptr [new_List]      # Create a new List using low level interface
List 12 -this $listptr      # Turn it into a List object named `12'

set 13 [List]               # Create a new list. The name of the list is in $13

List -this $listptr         # Turn $listptr into a Tcl object of the same name
```

Assuming you're not completely confused at this point, the best way to think of this is that there are really two different ways of representing an object. One approach is to simply use the pointer value as the name of an object. For example :

```
_100e8f8_p_List
```

The second approach is to allow you to pick a name for an object such as "foo". The different types of constructors are really just a mechanism for using either approach.

Invoking member functions

Member functions are invoked using the name of the object followed by the method name and any arguments. For example :

```
% List l
% l insert "Bob"
% l insert "Mary"
% l search "Dave"
0
%
```

Or if you let SWIG generate the name of the object, it works like this:

```
% set l [List]
% $l insert "Bob"           # Note $l contains the name of the object
% $l insert "Mary"
% $l search "Dave"
0
%
```

Deleting objects

Since objects are created by adding new Tcl commands, they can be deleted by simply renaming them. For example :

```
% rename l ""               # Destroy list object `l'
```

It is also possible to explicitly delete the object using the delete method. For example:

```
% l -delete
```

If applicable, SWIG will automatically call the corresponding C/C++ destructor when the object is destroyed.

Accessing member data

Member data of an object can be accessed using the `cget` method. The approach is quite similar to that used in [incr Tcl] and other Tcl extensions. For example :

```
% l cget -length          # Get the length of the list
13
```

The `cget` method currently only allows retrieval of one member at a time. Extracting multiple members will require repeated calls.

The member `-this` contains the pointer to the object that is compatible with other SWIG functions. Thus, the following call would be legal

```
% List l                # Create a new list object
% l insert Mike
% List_print [l cget -this]  # Print it out using low-level function
```

Changing member data

To change the value of member data, the `configure` method can be used. For example :

```
% l configure -length 10  # Change length to 10 (probably not a good idea, but
                           # possible).
```

In a structure such as the following :

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

you can change the value of all or some of the members as follows :

```
% v configure -x 3.5 -y 2 -z -1.0
```

The order of attributes does not matter.

Managing Object Ownership

By default, objects created from Tcl are owned by the Tcl interpreter and are automatically destroyed when the corresponding Tcl variable goes out of scope. However, sometimes it is necessary to change the ownership of an object managed by the interpreter. For instance, if you stored an object in another data structure, you might want Tcl to disown the object. Similarly, it may make sense for Tcl to acquire ownership of an object returned by C.

To handle object ownership, two object methods are available:

```
% obj -disown            # Release ownership
% obj -acquire            # Acquire ownership
```

When Tcl owns an object, it is released when the Tcl variable is destroyed. Otherwise, the Tcl variable is destroyed without calling the corresponding C/C++ destructor.

Relationship with pointers

The object oriented interface is mostly compatible with all of the functions that accept pointer values as arguments. Here are a couple of things to keep in mind :

- If you explicitly gave a name to an object, the pointer value can be retrieved using the ``cget -this'` method. The pointer value is what you should give to other SWIG generated functions if necessary.
- If you let SWIG generate the name of an object for you, then the name of the object is the same as the pointer value. This is the preferred approach.
- If you have a pointer value but it's not a Tcl object, you can turn it into one by calling the constructor with the ``-this'` option.

Here is a script that illustrates how these things work :

```
# Example 1 : Using a named object

List l                                # Create a new list
l insert Dave                         # Call some methods
l insert Jane
l insert Pat
List_print [l cget -this]             # Call a static method (which requires the pointer value)

# Example 2: Let SWIG pick a name

set l [List]                         # Create a new list
$l insert Dave                       # Call some methods
$l insert Jane
$l insert Pat
List_print $l                        # Call static method (name of object is same as pointer)

# Example 3: Already existing object
set l [new_List]                     # Create a raw object using low-level interface
List_insert $l Dave                  # Call some methods (using low-level functions)
List -this $l                        # Turn it into a Tcl object instead
$l insert Jane
$l insert Part
List_print $l                        # Call static method (uses pointer value as before).
```

Everything from this point on is out of date-- in progress

Examples

Accessing arrays

In some cases, C functions may involve arrays and other objects. In these instances, you may have to write helper functions to provide access. For example, suppose you have a C function like this :

```
// Add vector a+b -> c
void vector_add(double *a, double *b, double *c, int size);
```


SWIG is quite literal in its interpretation of `double *`—it is a pointer to a double. To provide access, a few helper functions can be written such as the following :

```
// SWIG helper functions for double arrays
%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;
}
%}
```

Using our C functions might work like this :

```
# Tcl code to create some arrays and add them

set a [new_double 200]
set b [new_double 200]
set c [new_double 200]

# Fill a and b with some values
for {set i 0} {$i < 200} {incr i 1} {
    set_double $a $i 0.0
    set_double $b $i $i
}

# Add them and store result in c
vector_add $a $b $c 200
```

The functions `get_double` and `set_double` can be used to access individual elements of an array. To convert from Tcl lists to C arrays, one could write a few functions in Tcl such as the following :

```
# Tcl Procedure to turn a list into a C array
proc Tcl2Array {l} {
    set len [llength $l]
    set a [new_double $len]
    set i 0
    foreach item $l {
        set_double $a $i $item
        incr i 1
    }
    return $a
}

# Tcl Procedure to turn a C array into a Tcl List
proc Array2Tcl {a size} {
    set l {}
    for {set i 0} {$i < size} {incr i 1} {
        lappend $l [get_double $a $i]
    }
    return $l
}
```

```
}
```

While not optimal, one could use these to turn a Tcl list into a C representation. The C representation could be used repeatedly in a variety of C functions without having to repeatedly convert from strings (Of course, if the Tcl list changed one would want to update the C version). Likewise, it is relatively simple to go back from C into Tcl. This is not the only way to manage arrays—typemaps can be used as well. The SWIG library file ``array.i'` also contains a variety of pre-written helper functions for managing different kinds of arrays.

Exception handling

The `%except` 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 exception handling 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 :

```
%except(tcl) {
    try {
        $function                // Gets substituted by actual function call
    }
    catch (RangeError) {
        interp->result = "Array index out-of-bounds";
        return TCL_ERROR;
    }
}
```

or in Tcl 8.0

```
%except(tcl8) {
    try {
        $function                // Gets substituted by actual function call
    }
    catch (RangeError) {
        Tcl_SetStringObj(tcl_result,"Array index out-of-bounds");
        return TCL_ERROR;
    }
}
```

When the C++ class throws a `RangeError` exception, our wrapper functions will catch it, turn it into a Tcl exception, and allow a graceful death as opposed to just having some sort of mysterious program crash. We are not limited to C++ exception handling. Please see the chapter on exception handling for more details on other possibilities, including a method for language-independent exception handling..

Typemaps

This section describes how SWIG's treatment of various C/C++ datatypes can be remapped using the `%typemap` directive. While not required, this section assumes some familiarity with Tcl's C API. The reader is advised to consult a Tcl book. A glance at the chapter on SWIG typemaps will also be useful.

What is a typemap?

A typemap is mechanism by which SWIG's processing of a particular C datatype can be changed. A simple typemap might look like this :

```
%module example

%typemap(tcl,in) int {
    $target = (int) atoi($source);
    printf("Received an integer : %d\n",$target);
}
...
extern int fact(int n);
```

Typemaps require a language name, method name, datatype, and conversion code. For Tcl, "tcl" should be used as the language name. For Tcl 8.0, "tcl8" should be used if you are using the native object interface. The "in" method in this example refers to an input argument of a function. The datatype ``int'` tells SWIG that we are remapping integers. The supplied code is used to convert from a Tcl string to the corresponding C datatype. Within the supporting C code, the variable `$source` contains the source data (a string in this case) and `$target` contains the destination of a conversion (a C local variable).

10 SWIG and Tcl

When the example is compiled into a Tcl module, it will operate as follows :

```
% fact 6
Received an integer : 6
720
%
```

A full discussion of typemaps can be found in the main SWIG users reference. We will primarily be concerned with Tcl typemaps here.

Tcl typemaps

The following typemap methods are available to Tcl modules :

`%typemap(tcl, in)` Converts a string to input function arguments

`%typemap(tcl, out)` Converts return value of a C function to a string

`%typemap(tcl, freearg)` Cleans up a function argument (if necessary)

`%typemap(tcl, argout)` Output argument processing

`%typemap(tcl, ret)` Cleanup of function return values

`%typemap(tcl, const)` Creation of Tcl constants

`%typemap(memberin)` Setting of C++ member data

`%typemap(memberout)` Return of C++ member data

`%typemap(tcl, check)` Check value of function arguments.

Typemap variables

The following variables may be used within the C code used in a typemap:

`$source` Source value of a conversion

`$target` Target of conversion (where the result should be stored)

`$type` C datatype being remapped

`$mangle` Mangled version of data (used for pointer type-checking)

`$value` Value of a constant (const typemap only)

`$arg` Original function argument (usually a string)

Name based type conversion

Typemaps are based both on the datatype and an optional name attached to a datatype. For example :

```
%module foo

// This typemap will be applied to all char ** function arguments
%typemap(tcl,in) char ** { ... }

// This typemap is applied only to char ** arguments named `argv'
%typemap(tcl,in) char **argv { ... }
```

In this example, two typemaps are applied to the `char **` datatype. However, the second typemap will only be applied to arguments named ``argv'`. A named typemap will always override an unnamed typemap.

Due to the name-based nature of typemaps, it is important to note that typemaps are independent of typedef declarations. For example :

```
%typemap(tcl, in) double {
    ... get a double ...
}
void foo(double);           // Uses the above typemap
typedef double Real;
void bar(Real);             // Does not use the above typemap (double != Real)
```

To get around this problem, the `%apply` directive can be used as follows :

```
%typemap(tcl,in) double {
    ... get a double ...
}
void foo(double);

typedef double Real;           // Uses typemap
%apply double { Real };       // Applies all "double" typemaps to Real.
void bar(Real);               // Now uses the same typemap.
```

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(tcl,in) char ** {
    int tempc;
    if (Tcl_SplitList(interp,$source,&tempc,&$target) == TCL_ERROR)
        return TCL_ERROR;
}

// This gives SWIG some cleanup code that will get called after the function call
%typemap(tcl,freearg) char ** {
    free((char *) $source);
}
```

10 SWIG and Tcl

```
}

// Return a char ** as a Tcl list
%typemap(tcl,out) char ** {
    int i = 0;
    while ($source[i]) {
        Tcl_AppendElement(interp,$source[i]);
        i++;
    }
}

// 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];
}

// A global variable
char *args[] = { "123", "54", "-2", "0", "NULL", 0 };

%}
#include tclsh.i
```

When compiled, we can use our functions as follows :

```
% print_args {John Guido Larry}
argv[0] = John
argv[1] = Guido
argv[2] = Larry
3
% puts [get_args]
Dave Mike Susan John Michelle
% puts [args_get]
123 54 -2 0 NULL
%
```

Perhaps the only tricky part of this example is the implicit memory allocation that is performed by the `Tcl_SplitList` function. To prevent a memory leak, we can use the SWIG "freearg" typemap to clean up the argument value after the function call is made. In this case, we simply free up the memory that `Tcl_SplitList` allocated for us.

Remapping constants

By default, SWIG installs C constants as Tcl read-only variables. Unfortunately, this has the undesirable side effect that constants need to be declared as "global" when used in subroutines. For example :

```

proc clearscreen { } {
    global GL_COLOR_BUFFER_BIT
    glClear $GL_COLOR_BUFFER_BIT
}

```

If you have hundreds of functions however, this quickly gets annoying. Here's a fix using hash tables and SWIG typemaps :

```

// Declare some Tcl hash table variables
%{
static Tcl_HashTable  constTable;      /* Hash table          */
static int            *swigconst;      /* Temporary variable */
static Tcl_HashEntry  *entryPtr;       /* Hash entry          */
static int            dummy;           /* dummy value         */
}%

// Initialize the hash table (This goes in the initialization function)

%init %{
    Tcl_InitHashTable(&constTable,TCL_STRING_KEYS);
}%

// A Typemap for creating constant values
// $source = the value of the constant
// $target = the name of the constant

%typemap(tcl,const) int, unsigned int, long, unsigned long {
    entryPtr = Tcl_CreateHashEntry(&constTable,$target,&dummy);
    swigconst = (int *) malloc(sizeof(int));
    *swigconst = $source;
    Tcl_SetHashValue(entryPtr, swigconst);
    /* Make it so constants can also be used as variables */
    Tcl_LinkVar(interp,$target, (char *) swigconst, TCL_LINK_INT | TCL_LINK_READ_ONLY);
};

// Now change integer handling to look for names in addition to values
%typemap(tcl,in) int, unsigned int, long, unsigned long {
    Tcl_HashEntry *entryPtr;
    entryPtr = Tcl_FindHashEntry(&constTable,$source);
    if (entryPtr) {
        $target = ($type) (*(int *) Tcl_GetHashValue(entryPtr));
    } else {
        $target = ($type) atoi($source);
    }
}

```

In our Tcl code, we can now access constants by name without using the "global" keyword as follows :

```

proc clearscreen { } {
    glClear GL_COLOR_BUFFER_BIT
}

```

Returning values in arguments

The "argout" typemap can be used to return a value originating from a function argument. For example :

10 SWIG and Tcl

```
// A typemap defining how to return an argument by appending it to the result
%typemap(tcl,argout) double *outvalue {
    char dtemp[TCL_DOUBLE_SPACE];
    Tcl_PrintDouble(interp,*( $source),dtemp);
    Tcl_AppendElement(interp, dtemp);
}

// A typemap telling SWIG to ignore an argument for input
// However, we still need to pass a pointer to the C function
%typemap(tcl,ignore) double *outvalue {
    static double temp;          /* A temporary holding place */
    $target = &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 `"ignore"` typemap 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
%
```

An alternative approach to this is to return values in a Tcl variable as follows :

```
%typemap(tcl,argout) double *outvalue {
    char temp[TCL_DOUBLE_SPACE];
    Tcl_PrintDouble(interp,*( $source),dtemp);
    Tcl_SetVar(interp,$arg,temp,0);
}
%typemap(tcl,in) double *outvalue {
    static double temp;
    $target = &temp;
}
```

Our Tcl script can now do the following :

```
% set status [mypow 2 3 a]
% puts $status
0
% puts $a
8.0
%
```

Here, we have passed the name of a Tcl variable to our C wrapper function which then places the return value in that variable. This is now very close to the way in which a C function calling this function would work.

Mapping C structures into Tcl Lists

Suppose you have a C structure like this :

```
typedef struct {
    char login[16];           /* Login ID */
    int uid;                  /* User ID */
    int gid;                  /* Group ID */
    char name[32];            /* User name */
    char home[256];           /* Home directory */
} User;
```

By default, SWIG will simply treat all occurrences of "User" as a pointer. Thus, functions like this :

```
extern void add_user(User u);
extern User *lookup_user(char *name);
```

will work, but they will be weird. In fact, they may not work at all unless you write helper functions to create users and extract data. A typemap can be used to fix this problem however. For example :

```
// This works for both "User" and "User *"
%typemap(tcl,in) User * {
    int tempc;
    char **tempa;
    static User temp;
    if (Tcl_SplitList(interp,$source,&tempc,&tempa) == TCL_ERROR) return TCL_ERROR;
    if (tempc != 5) {
        free((char *) tempa);
        interp->result = "Not a valid User record";
        return TCL_ERROR;
    }
    /* Split out the different fields */
    strncpy(temp.login,tempa[0],16);
    temp.uid = atoi(tempa[1]);
    temp.gid = atoi(tempa[2]);
    strncpy(temp.name,tempa[3],32);
    strncpy(temp.home,tempa[4],256);
    $target = &temp;
    free((char *) tempa);
}

// Describe how we want to return a user record
%typemap(tcl,out) User * {
    char temp[20];
    if ($source) {
        Tcl_AppendElement(interp,$source->login);
        sprintf(temp,"%d",$source->uid);
        Tcl_AppendElement(interp,temp);
        sprintf(temp,"%d",$source->gid);
        Tcl_AppendElement(interp,temp);
        Tcl_AppendElement(interp,$source->name);
        Tcl_AppendElement(interp,$source->home);
    }
}
```

10 SWIG and Tcl

These function marshall Tcl lists to and from our User data structure. This allows a more natural implementation that we can use as follows :

```
% add_user {beazley 500 500 "Dave Beazley" "/home/beazley"}
% lookup_user beazley
beazley 500 500 {Dave Beazley} /home/beazley
```

This is a much cleaner interface (although at the cost of some performance). The only caution I offer is that the pointer view of the world is pervasive throughout SWIG. Remapping complex datatypes like this will usually work, but every now and then you might find that it breaks. For example, if we needed to manipulate arrays of Users (also mapped as a "User *"), the typemaps defined here would break down and something else would be needed. Changing the representation in this manner may also break the object-oriented interface.

Useful functions

The following tables provide some functions that may be useful in writing Tcl typemaps. Both Tcl 7.x and Tcl 8.x are covered. For Tcl 7.x, everything is a string so the interface is relatively simple. For Tcl 8, everything is now a Tcl object so a more precise set of functions is required. Given the alpha-release status of Tcl 8, the functions described here may change in future releases.

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 *
```

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);
```

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, == TCL_ERROR)
        return TCL_ERROR;
    $1 = ($1_ltype) temp;
}

%typemap(out) int, short, long {
    Tcl_SetIntObj($result, (int) $1);
}
```

Floating point conversion

```
%typemap(in) float, double {
    double temp;
    if (Tcl_GetDoubleFromObj(interp, $input, == TCL_ERROR)
        return TCL_ERROR;
    $1 = ($1_ltype) temp;
}

%typemap(out) float, double {
    Tcl_SetDoubleObj($result, $1);
}
```

String Conversion

```
%typemap(in) char * {
    int len;
    $1 = Tcl_GetStringFromObj(interp,
    }

%typemap(out) char * {
    Tcl_SetStringObj($result, $1);
}
```

Pointer handling

REWRITE THIS!

SWIG pointers are mapped into Python strings containing the hexadecimal value and type. The following functions can be used to create and read pointer values.

These functions can be used in typemaps as well. For example, the following typemap makes an argument of "char *buffer" accept a pointer instead of a NULL-terminated ASCII string.

```
%typemap(tcl,in) char *buffer {
    if (SWIG_GetPtr($source, (void **) &$target, "$mangle")) {
        Tcl_SetResult(interp,"Type error. Not a pointer", TCL_STATIC);
        return TCL_ERROR;
    }
}
```

Note that the `$mangle` variable generates the type string associated with the datatype used in the `typemap`.

By now you hopefully have the idea that `typemaps` are a powerful mechanism for building more specialized applications. While writing `typemaps` can be technical, many have already been written for you. See the SWIG library reference for more information.

Configuration management with SWIG

After you start to work with Tcl for awhile, you suddenly realize that there are an unimaginable number of extensions, tools, and other packages. To make matters worse, there are about 20 billion different versions of Tcl, not all of which are compatible with each extension (this is to make life interesting of course).

While SWIG is certainly not a magical solution to the configuration management problem, it can help out a lot in a number of key areas :

- SWIG generated code can be used with all versions of Tcl/Tk newer than 7.3/3.6. This includes the Tcl Netscape Plugin and Tcl 8.0a2.
- The SWIG library mechanism can be used to manage various code fragments and initialization functions.
- SWIG generated code usually requires no modification so it is relatively easy to switch between different Tcl versions as necessary or upgrade to a newer version when the time comes (of course, the Sun Tcl/Tk team might have changed other things to keep you occupied)

Writing a main program and `Tcl_AppInit()`

The traditional method of creating a new Tcl extension required a programmer to write a special function called `Tcl_AppInit()` that would initialize your extension and start the Tcl interpreter. A typical `Tcl_AppInit()` function looks like the following :

```
/* main.c */
#include <tcl.h>

main(int argc, char *argv[]) {
    Tcl_Main(argc,argv);
    exit(0);
}

int Tcl_AppInit(Tcl_Interp *interp) {
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    /* Initialize your extension */
    if (Your_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
}
```

```

        tcl_RcFileName = "~/myapp.tcl";
        return TCL_OK;
    }

```

While relatively simple to write, there are tons of problems with doing this. First, each extension that you use typically has their own `Tcl_AppInit()` function. This forces you to write a special one to initialize everything by hand. Secondly, the process of writing a main program and initializing the interpreter varies between different versions of Tcl and different platforms. For example, in Tcl 7.4, the variable `"tcl_RcFileName"` is a C variable while in Tcl 7.5 and newer versions it's a Tcl variable instead. Similarly, the `Tcl_AppInit` function written for a Unix machine might not compile correctly on a Mac or Windows machine.

In SWIG, it is almost never necessary to write a `Tcl_AppInit()` function because this is now done by SWIG library files such as `tclsh.i` or `wish.i`. To give a better idea of what these files do, here's the code from the SWIG `tclsh.i` file which is roughly comparable to the above code

```

// tclsh.i : SWIG library file for rebuilding tclsh
%{

/* A Tcl_AppInit() function that lets you build a new copy
 * of tclsh.
 *
 * The macro SWIG_init contains the name of the initialization
 * function in the wrapper file.
 */

#ifndef SWIG_RcFileName
char *SWIG_RcFileName = "~/myapprc";
#endif

int Tcl_AppInit(Tcl_Interp *interp){

    if (Tcl_Init(interp) == TCL_ERROR)
        return TCL_ERROR;

    /* Now initialize our functions */
    if (SWIG_init(interp) == TCL_ERROR)
        return TCL_ERROR;

    #if TCL_MAJOR_VERSION > 7 || (TCL_MAJOR_VERSION == 7 && TCL_MINOR_VERSION >= 5)
        Tcl_SetVar(interp, "tcl_rcFileName", SWIG_RcFileName, TCL_GLOBAL_ONLY);
    #else
        tcl_RcFileName = SWIG_RcFileName;
    #endif
    return TCL_OK;
}

#if TCL_MAJOR_VERSION > 7 || (TCL_MAJOR_VERSION == 7 && TCL_MINOR_VERSION >= 4)
int main(int argc, char **argv) {
    Tcl_Main(argc, argv, Tcl_AppInit);
    return(0);
}
#else
extern int main();
#endif

%}

```

10 SWIG and Tcl

This file is essentially the same as a normal `Tcl_AppInit()` function except that it supports a variety of Tcl versions. When included into an interface file, the symbol `SWIG_init` contains the actual name of the initialization function (This symbol is defined by SWIG when it creates the wrapper code). Similarly, a startup file can be defined by simply defining the symbol `SWIG_RcFileName`. Thus, a typical interface file might look like this :

```
%module graph
%{
#include "graph.h"
#define SWIG_RcFileName "graph.tcl"
%}

#include tclsh.i

... declarations ...
```

By including the `tclsh.i`, you automatically get a `Tcl_AppInit()` function. A variety of library files are also available. `wish.i` can be used to build a new wish executable, `expect.i` contains the main program for Expect, and `ish.i`, `itclsh.i`, `iwish.i`, and `itkwish.i` contain initializations for various incarnations of [incr Tcl].

Creating a new package initialization library

If a particular Tcl extension requires special initialization, you can create a special SWIG library file to initialize it. For example, a library file to extend Expect looks like the following :

```
// expect.i : SWIG Library file for Expect
%{

/* main.c - main() and some logging routines for expect

Written by: Don Libes, NIST, 2/6/90

Design and implementation of this program was paid for by U.S. tax
dollars. Therefore it is public domain. However, the author and NIST
would appreciate credit if this program or parts of it are used.
*/

#include "expect_cf.h"
#include <stdio.h>
#include INCLUDE_TCL
#include "expect_tcl.h"

void
main(argc, argv)
int argc;
char *argv[];
{
    int rc = 0;
    Tcl_Interp *interp = Tcl_CreateInterp();
    int SWIG_init(Tcl_Interp *);

    if (Tcl_Init(interp) == TCL_ERROR) {
        fprintf(stderr, "Tcl_Init failed: %s\n", interp->result);
        exit(1);
    }
}
```

```

    }
    if (Exp_Init(interp) == TCL_ERROR) {
        fprintf(stderr,"Exp_Init failed: %s\n",interp->result);
        exit(1);
    }

    /* SWIG initialization. --- 2/11/96 */
    if (SWIG_init(interp) == TCL_ERROR) {
        fprintf(stderr,"SWIG initialization failed: %s\n", interp->result);
        exit(1);
    }

    exp_parse_argv(interp,argc,argv);
    /* become interactive if requested or "nothing to do" */
    if (exp_interactive)
        (void) exp_interpreter(interp);
    else if (exp_cmdfile)
        rc = exp_interpret_cmdfile(interp,exp_cmdfile);
    else if (exp_cmdfilename)
        rc = exp_interpret_cmdfilename(interp,exp_cmdfilename);

    /* assert(exp_cmdlinecmds != 0) */
    exp_exit(interp,rc);
    /*NOTREACHED*/
}
%}

```

In the event that you need to write a new library file such as this, the process usually isn't too difficult. Start by grabbing the original `Tcl_AppInit()` function for the package. Enclose it in a `%{ , %}` block. Now add a line that makes a call to `SWIG_init()`. This will automatically resolve to the real initialization function when compiled.

Combining Tcl/Tk Extensions

A slightly different problem concerns the mixing of various extensions. Most extensions don't require any special initialization other than calling their initialization function. To do this, we also use SWIG library mechanism. For example :

```

// blt.i : SWIG library file for initializing the BLT extension
%{
#ifdef __cplusplus
extern "C" {
#endif
extern int Blt_Init(Tcl_Interp *);
#ifdef __cplusplus
}
#endif
%}
%init %{
    if (Blt_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
%}

// tix.i : SWIG library file for initializing the Tix extension
%{
#ifdef __cplusplus

```

10 SWIG and Tcl

```
extern "C" {
#ifdef
extern int Tix_Init(Tcl_Interp *);
#endif
#ifdef __cplusplus
}
#endif
%}
%init %{
    if (Tix_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
%}
```

Both files declare the proper initialization function (to be C++ friendly, this should be done using `extern "C"`). A call to the initialization function is then placed inside a `%init %{ ... %}` block.

To use our library files and build a new version of wish, we might now do the following :

```
// mywish.i : wish with a bunch of stuff added to it
#include wish.i
#include blt.i
#include tix.i

... additional declarations ...
```

Of course, the really cool part about all of this is that the file ``mywish.i'` can itself, serve as a library file. Thus, when building various versions of Tcl, we can place everything we want to use a special file and use it in all of our other interface files :

```
// interface.i
%module mymodule

#include mywish.i           // Build our version of Tcl with extensions

... C declarations ...
```

or we can grab it on the command line :

```
unix > swig -tcl -lmywish.i interface.i
```

Limitations to this approach

This interface generation approach is limited by the compatibility of each extension you use. If any one extension is incompatible with the version of Tcl you are using, you may be out of luck. It is also critical to pay careful attention to libraries and include files. An extension library compiled against an older version of Tcl may fail when linked with a newer version.

Dynamic loading

Newer versions of Tcl support dynamic loading. With dynamic loading, you compile each extension into a separate module that can be loaded at run time. This simplifies a number of compilation and extension building

problems at the expense of creating new ones. Most notably, the dynamic loading process varies widely between machines and is not even supported in some cases. It also does not work well with C++ programs that use static constructors. Modules linked with older versions of Tcl may not work with newer versions as well (although SWIG only really uses the basic Tcl C interface). As a result, I usually find myself using both dynamic and static linking as appropriate.

Turning a SWIG module into a Tcl Package.

Tcl 7.4 introduced the idea of an extension package. By default, SWIG does not create "packages", but it is relatively easy to do. To make a C extension into a Tcl package, you need to provide a call to `Tcl_PkgProvide()` in the module initialization function. This can be done in an interface file as follows :

```
%init {%
    Tcl_PkgProvide(interp, "example", "0.0");
%}
```

Where "example" is the name of the package and "0.0" is the version of the package.

Next, 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.

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).

Shadow classes

A similar approach can be applied to 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
}

```

10 SWIG and Tcl

```
#
return $handle
}

proc deleteObject {objectType handle name element op} {
    #
    # Check that the object handle has a reasonable form
    #
    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.

SWIG 1.3 – Last Modified : January 31, 2002

SWIG and Java

- [Preliminaries](#)
- [Building Java Extensions under Windows 95/NT](#)
- [The low-level Java/C interface](#)
- [Java shadow classes](#)
- [Examples](#)
- [Exception handling](#)
- [Remapping C datatypes with typemaps](#)
- [The gory details of shadow classes](#)
- [Java pragmas](#)
- [Dynamic linking problems](#)
- [Tips](#)
- [Known bugs](#)

This chapter describes SWIG's support of Java. Java is one of the latest modules to be added to SWIG. The other SWIG modules are primarily scripting languages. Using Java has the advantage over scripting languages of being type safe. The 100% Pure Java effort is a commendable concept however, in the real world programmers 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. With this Java extension to SWIG it is 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 Java functions and creates C/C++ function prototypes.

Preliminaries

SWIG 1.1 works with JDK 1.1 and higher. Given the choice, you should use the latest version of Sun's JDK. The SWIG Java module is known to work on Solaris, the various flavours of Windows including cygwin and Linux using Sun's JDK. It is also known to work on vxWorks using their PJava 3.1. 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 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 the `make check` from the SWIG root directory after installing SWIG.

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 is more than likely to support this.

Running SWIG

The basics of getting a SWIG Java module up and running can be seen from one of SWIG's example Makefiles, but is also described here. To build a Java module, run SWIG using the `-java` option. Enabling shadow classes `-shadow` is also recommended:

```
%swig -java -shadow example.i
```

This will produce 2 files. The file `example_wrap.c` contains all of the C code needed to build a Java module. To build a Java module, you will need to compile the file `example_wrap.c` to create a shared library. When shadow classes are enabled, SWIG may also produce many `.java` files, but this is described later.

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

<code>-jnic</code>	use c syntax for JNI calls (default depends on <code>-c++</code> flag)
<code>-jnicpp</code>	use C++ syntax for JNI calls (default depends on <code>-c++</code> flag)
<code>-module <module></code>	set name of module
<code>-package <java package></code>	set the name of the package for the generated classes
<code>-shadow</code>	generate shadow classes
<code>-nofinalize</code>	do not generate finalizer methods in shadow classes

Their use will become clearer by the time you have finished reading this section on SWIG and Java.

Getting the right header files

In order to compile, you need to locate the "jni.h" and "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.

Compiling a dynamic module

The JNI code exists in a dynamic module or shared object and gets loaded by the JVM. To build a shared object file, you need to compile your module in a manner similar to the following (shown for Solaris):

```
$ swig -java -shadow example.i
$ gcc -c example_wrap.c -I/usr/java/include -I/usr/java/include/solaris
$ ld -G example_wrap.o -o libexample.so
```

Unfortunately, the process of building a shared object file varies on every single machine so you may need to read up on the man pages for your C compiler and linker.

When building a dynamic module, the name of the output file is important. If the name of your SWIG module is "example", the name of the corresponding object 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.

Using your module

To use your module in Java, simply use Java's `import` command and `System.loadLibrary` method in a Java class:

```
// main.java
import example;

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
$
```

Compilation problems and compiling with C++

For the most part, compiling a Java module is straightforward, but there are a number of potential problems :

- In order to build C++ modules, you may need to link with the C++ compile using a command like ``c++ -shared example_wrap.o example.o -o libexample.so'`
- If building a dynamic C++ module using `g++`, you may also need to link against `libgcc.a`, `libg++.a`, and `libstdc++.a` libraries.
- Make sure you are using the version of JDK header files matches the version of Java that you are running.

Building Java Extensions under Windows 95/NT

Building a SWIG extension to Java under Windows 95/NT 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 to build extensions, you will need to have a JDK installed on your machine in order to read the JNI header files.

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 `example_wrap.cxx`. Don't worry if the wrapper file doesn't exist yet—Developer 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 -shadow -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.

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](#).

Using NMAKE

Alternatively, SWIG extensions can be built by writing a Makefile for NMAKE. 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 a Java extension

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 -shadow -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 extension, run `NMAKE` (you may need to run `vcvars32` first). This is a pretty simplistic Makefile, but hopefully its enough to get you started.

The low-level Java/C interface

The SWIG Java module is based upon a basic low-level interface that provides access to C functions, variables, constants, and C++ classes. This low-level interface is not the recommended way to use SWIG with Java; shadow classes are the recommended way. This low-level interface is used by the shadow class interface so it is used behind the scenes.

Modules and the module class

The SWIG `%module` directive specifies the name of the Java module. If you specified ``%module example'`, then everything found in a SWIG interface file will be contained within the Java module class which in this case will be called ``example'`. In other words the module class contains all the C/C++ functions and accessor functions to variables that have been wrapped by SWIG. Make sure you don't use the same name as a Java keyword for your module name else the Java output will not compile.

Functions

C/C++ functions are mapped directly into a matching method in a Java class named after the module name. For example :

```

%module example
extern int fact(int n);

```

Will produce the following JNI c function:

```

JNIEXPORT jint JNICALL Java_example_fact(JNIEnv *jenv, jclass jcls, jint jarg0) {

```

SWIG and Java

This will in turn call the desired `fact` function. The JNI naming mangling isn't pretty, but you can see from the above that the native function call is expecting a class 'example'. SWIG outputs the JNI Java code into the example class:

```
public class example {
    public final static native int fact(int jarg0);
}
```

It can be used as follows from Java:

```
System.out.println(example.fact(4));
```

Variable Linking

SWIG provides access to C/C++ global variables through the class named after the module name, in other words all global variables are wrapped into the module class. Java does not allow the overriding of the dot operator so all variables are accessed through getters and setters. For example:

```
/* SWIG interface file with global variables */
%module example
...
extern int My_variable;
...
```

Now in Java :

```
// Print out the value of a c global variable
System.out.println("My Variable = " + example.get_My_variable());

//Set the value of a C global variable
example.set_My_variable(100);
```

The value returned by the getter will always be up to date even if the value is changed in c. The setter will not be generated if the variable is a c 'const'.

Enums

SWIG will wrap enumerations. They appear as public final static Java variables. The Java enum names match the C/C++ enum names. For example, the following C enum:

```
enum color { RED, BLUE, GREEN };
```

will be wrapped with the following Java:

```
public class example {
... maybe some other functions ...
    public final static native int get_RED();
    public final static native int get_BLUE();
    public final static native int get_GREEN();
    // enums and constants
    public final static int RED = get_RED();
    public final static int BLUE = get_BLUE();
```

```
    public final static int GREEN = get_GREEN();
}
```

Constants

C/C++ constants (from a `#define`) are wrapped by public final static Java variables. These constants are given the same name as the corresponding C constant. For example, the following C:

```
#define    ICONST    42
#define    FCONST    2.1828
#define    CCONST    'x'
#define    SCONST    "Hello World"
```

will be wrapped with the following Java:

```
public final static int ICONST = 42;
public final static double FCONST = 2.1828;
public final static String CCONST = "x";
public final static String SCONST = "Hello World";
```

Pointers

All c pointers are treated as Java longs in the low-level interface. For example:

```
int* pointer_fn(short* a, int* b, long* c);
```

will produce the following Java function:

```
public final static native long pointer_fn(long jarg0, long jarg1, long jarg2);
```

Unlike other language modules, no SWIG pointer library exists yet for the Java module. Until this is written it does restrict the functions that can be usefully used from Java. In the above `pointer_fn`, any long representing a c pointer can be passed to other c functions, but any value that the pointer points to cannot be accessed from Java without writing and using some accessor functions, for example:

```
/* c accessor function for reading return values that are int pointers */
int getIntValue(int *ptr) {return *ptr;};

int* pointer_fn();
```

The return value can then be read from Java, for example:

```
long integerPtr = example.pointer_fn();
System.out.println("return value=" + example.getIntValue(integerPtr));
```

Pointers that are returned in one of the parameter arguments are more difficult and other workarounds are necessary. One method is to write a c function which returns the desired value after calling the appropriate function. However, as the section on shadow classes demonstrates, the situation is a lot better when using shadow classes.

Another alternative is to use the `typemaps.i` library. This file has examples to demonstrate, but essentially if you

use the typemaps in this library you can use real values from Java where a pointer is required in c. When the pointer is an output from the c function, a Java array is used. The value that the pointer is pointing to is placed into the array which can be read by the calling function.

Structures

The low-level SWIG interface only provides a simple interface to C structures. For example :

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

This will be wrapped using native Java functions, where the first argument holds the pointer to an instance of the Vector struct. Example assuming the module name is 'example':

```
public class example {
    public final static native void    set_Vector_x(long jarg0, double jarg1);
    public final static native double get_Vector_x(long jarg0);
    public final static native void    set_Vector_y(long jarg0, double jarg1);
    public final static native double get_Vector_y(long jarg0);
    public final static native void    set_Vector_z(long jarg0, double jarg1);
    public final static native double get_Vector_z(long jarg0);
}
```

These functions are then used in the resulting Java interface. For example:

```
// v is a long holding the c pointer to a Vector that got created somehow
example.set_Vector_x(v, 7.8);
System.out.println("x=" + example.get_Vector_x(v));
```

When executed will display:

```
x=7.8
```

Similar access is provided for unions and the data members of C++ classes.

C++ Classes

C++ classes are handled 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, will produce the following JNI c code to access the class (assuming the module is set to 'example'):

```
JNIEXPORT jlong JNICALL Java_example_new_lList(JNIEnv *jenv, jclass jcls);
JNIEXPORT void JNICALL Java_example_delete_lList(JNIEnv *jenv, jclass jcls, jlong jarg0);
JNIEXPORT jint JNICALL Java_example_List_lsearch(JNIEnv *jenv, jclass jcls, jlong jarg0, jst
JNIEXPORT void JNICALL Java_example_List_linsert(JNIEnv *jenv, jclass jcls, jlong jarg0, jst
JNIEXPORT void JNICALL Java_example_List_lremove(JNIEnv *jenv, jclass jcls, jlong jarg0, jst
JNIEXPORT jstring JNICALL Java_example_List_lget(JNIEnv *jenv, jclass jcls, jlong jarg0, jin
JNIEXPORT void JNICALL Java_example_set_lList_llength(JNIEnv *jenv, jclass jcls, jlong jarg0
JNIEXPORT jint JNICALL Java_example_get_lList_llength(JNIEnv *jenv, jclass jcls, jlong jarg0
JNIEXPORT void JNICALL Java_example_List_lprint(JNIEnv *jenv, jclass jcls, jlong jarg0);
```

where jarg0 is the 'this' pointer in non-static functions. The JNI specification requires a c interface. The following JNI Java functions are also produced for access from Java:

```
public class example {
    public final static native long new_List();
    public final static native void delete_List(long jarg0);
    public final static native int List_search(long jarg0, String jarg1);
    public final static native void List_insert(long jarg0, String jarg1);
    public final static native void List_remove(long jarg0, String jarg1);
    public final static native String List_get(long jarg0, int jarg1);
    public final static native void List_length_set(long jarg0, int jarg1);
    public final static native int List_length_get(long jarg0);
    public final static native void List_print(long jarg0);
}
```

From Java, these functions can be used to access the C++ class:

```
long l = example.new_List();
example.List_insert(l, "Ale");
example.List_insert(l, "Stout");
example.List_insert(l, "Lager");
example.List_print(l);
int len = example.get_List_length(l);
System.out.println(len);
```

When executed might display:

```
Lager
Stout
Ale
3
```

While somewhat primitive, the low-level SWIG interface provides direct and flexible access to C++ objects. As it turns out, a more elegant method of accessing structures and classes is available using shadow classes.

Java shadow classes

The low-level interface generated by SWIG provides access to C structures and C++ classes, but it doesn't look much like a class that might be created in Java. However, it is possible to indirectly use the low-level C interface to write a Java class that looks like the original C++ class. In this case, the Java class is said to "shadow" the C++ class. That is, it behaves like the original class, but is really just a wrapper around a C++ class. The Java class is

sometimes called a proxy class.

A simple example

For our earlier List class, a Java shadow class could be written by hand like this :

```
public class List {
    protected long _cPtr;
    protected boolean _cMemOwn;

    public long getCPtr() {
        return _cPtr;
    }

    public List() {
        _cPtr = example.new_List();
        _cMemOwn = true;
    }

    protected void finalize() {
        _delete();
    }

    public void _delete() {
        if(_cPtr!=0 & _cMemOwn) {
            example.delete_List(_cPtr);
            _cPtr = 0;
        }
    }

    public int search(String item) {
        return example.List_search(_cPtr, item);
    }

    public void insert(String item) {
        example.List_insert(_cPtr, item);
    }

    public void remove(String item) {
        example.List_remove(_cPtr, item);
    }

    public String get(int n) {
        return example.List_get(_cPtr, n);
    }

    public void setLength(int value) {
        example.set_List_length(_cPtr, value);
    }

    public int getLength() {
        return example.get_List_length(_cPtr);
    }

    public static void print(List l) {
        example.List_print(l.getCPtr());
    }
}
```



```
}
```

Which is roughly what SWIG outputs. When used in Java, we can use the class as follows:

```
List l = new List();
l.insert("Ale");
l.insert("Stout");
l.insert("Lager");
List.print(l);
int len = l.getLength();
System.out.println(len);
```

Obviously, this is a much nicer interface than before, it only required a small amount of Java coding, it is type-safe and fits in with the Java programming paradigm.

Why write shadow classes in Java?

While one could wrap C/C++ objects directly into Java as new Java types, this approach has a number of problems. First, as the C/C++ code gets complicated, the resulting wrapper code starts to become extremely ugly.

By writing shadow classes in Java instead of C, the classes become real Java classes that can be easily used as base-classes in an inheritance hierarchy or for other applications. Writing the shadow classes in Java also greatly simplifies coding complexity as writing them in Java is much easier than trying to accomplish the same thing in C. Finally, by writing shadow classes in Java, they are easy to modify and can be changed without ever recompiling any of the C code. The downside to using shadow classes over the simple interface is a slight performance degradation—a concern for some users.

Automated shadow class generation

SWIG automatically generates shadow classes when you use the `-shadow` option:

```
swig -java -shadow interface.i
```

This will create the following files:

```
interface_wrap.c
interface.java
```

plus other .java files corresponding to each shadow class

The file `interface_wrap.c` contains the normal SWIG C JNI wrappers. The file `interface.java` contains the Java code corresponding to the Java native functions. The name of this file will be the same as specified by the `%module` directive in the SWIG interface file. These two files are produced whether or not the `-shadow` option is passed to SWIG. There will then be a `.java` file for each shadow class when `-shadow` is used. Note that if `-c++` is passed to SWIG for wrapping C++ code, then a `interface_wrap.cxx` file replaces the `interface_wrap.c` file.

Compiling modules with shadow classes

No changes need to be made to the compilation process when using shadow classes.

Where to go for more information

Shadow classes turn out to be so useful that they are used almost all of the time with SWIG. All of the examples presented here will assume that shadow classes have been enabled. The precise implementation of shadow classes is described at the end of this chapter.

Examples

The directory Examples/java has a number of examples. Looking at these is a good way to learn how the SWIG Java extension works. The Examples/index.html in the parent directory contains the SWIG Examples Documentation and is a useful starting point. The following sections also describe plenty of examples.

Exception handling

The SWIG `%exception` directive can be used to create a user-definable exception handler which can be used to convert errors in your C/C++ program into Java exceptions. The chapter on exception handling 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();
        }
    }
};

```

The functions associated with this class can throw a C++ range exception for an out-of-bounds array access. We can catch the C++ exception and rethrow it as a Java exception by specifying the following in an interface file :

```

%exception {
    try {
        $action
    }
    catch (RangeError) {
        jclass clazz = jenv->FindClass("java/lang/Exception");
        jenv->ThrowNew(clazz, "Range error");
        return $null;
    }
}

```

What the above does is define swig exception handlers for two types of functions; namely those that have a void return and those that do not. Both are needed to cover all types of JNI functions. The code excerpts above are inserted into all the JNI functions. The `$action` call is replaced by the C/C++ code being executed by the wrapper, for example the `setitem` call. Another special variable, `$null`, is useful for typemaps that will be used in JNI functions returning all return types. See the section on [Typemap variables](#) for further explanation.

The above uses the C++ JNI calling syntax as opposed to the C calling syntax and so will not compile as C. It is however possible to write JNI calls in [typemaps for both C and C++ compilation](#). The `ThrowNew()` JNI function must take a class derived from `java.lang.Exception`. The `%exception` typemap above could be tailored to individual needs.

When the C++ class throws a `RangeError` exception, our wrapper functions will catch it, turn it into a Java exception, and allow a graceful death as opposed to having some sort of mysterious JVM crash. Since SWIG's exception handling is user-definable, we are not limited to C++ exception handling. Please see the chapter on exception handling for more details and using the `exception.i` library for writing language-independent exception handlers.

See the chapter on [Exception Handling](#) for further examples on exceptions and how `%exception` can be targeted for use in a particular function.

If we use the following code:

```

final int SIZE=5;
DoubleArray arr = new DoubleArray(SIZE);
for (int i=0; i<SIZE; i++)
    arr.setitem(i, (double)i);
for (int i=0; i<SIZE+1; i++) //Note the array over bounds
    System.out.println(i + " " + arr.getitem(i));

```

Something similar to the following will be output when it is run:

```

0 0.0
1 1.0

```

```

2 2.0
3 3.0
4 4.0
Exception in thread "main" java.lang.Exception: Range error
    at example.DoubleArray_getitem(Native Method)
    at example.DoubleArray_getitem(Compiled Code)
    at DoubleArray.getitem(Compiled Code)
    at main.main(Compiled Code)

```

Remapping C datatypes with typemaps

This section describes how SWIG's treatment of various C/C++ datatypes can be remapped using the SWIG `%typemap` directive. While not required, this section assumes some familiarity with the JNI. The reader is advised to be familiar with the chapter on SWIG typemaps. Also it is best to consult JNI documentation either online at [Sun's Java web site](http://www.sun.com/javase/6/docs/api/java/lang/package-summary.html) or a good book on the JNI. 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.

Default type mapping

The following table lists the default type mapping from Java to C/C++.

C/C++ type	Java type	JNI type
char	byte	jbyte
unsigned char	short	jshort
short	short	jshort
unsigned short	int	jint
int	int	jint
unsigned int	long	jlong
long	int	jint
unsigned long	long	jlong
long long	long	jlong
unsigned long long	java.math.BigInteger	jobject
float	float	jfloat
double	double	jdouble
bool	boolean	jboolean
void	void	void

Arrays are implemented using the same mappings, for example a c array, `char[SIZE]`, is mapped to a Java array, `byte[SIZE]`.

When SWIG is being used without shadow classes, longs are used for all pointers. All complex types (C/C++ structs and classes) are accessible using a Java long which holds a pointer to the underlying C/C++ object. Arrays

of complex types are mapped as arrays of pointers to the type, that is a Java long[].

The output is different when SWIG is being used to generate shadow classes. Primitive types and pointers to these types remain the same; so a Java long is used for pointers to primitive types. However, for complex types, the shadow classes use a Java class which wraps the struct/class instead of a Java long. This applies to both argument parameters and return types that are complex types or pointers to complex types. Arrays of complex types turn into arrays of Java classes rather than arrays of longs.

For example, given the following C++ function call:

```
void AClass::func(int a, int* b, SomeClass c, SomeClass* d, SomeClass e, SomeClass f[10]);
```

The non shadow access from Java is shown below, where the first parameter, ptr, is a long containing the pointer to an object of type AClass:

```
public final static native void AClass_func(long ptr, int a, long b, long c, long d, long e
```

The Java shadow class, AClass, will contain the following function:

```
public void func(int a, long b, SomeClass c, SomeClass d, SomeClass e, SomeClass f[]) {...}
```

The mappings for C int and C long are appropriate for 32 bit applications which are used in the current 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 C values from Java.

For future 64 bit JVMs you may have to change the C long, but probably not C int 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.

What is a typemap?

A typemap is mechanism by which SWIG's processing of a particular C datatype can be overridden. A simple typemap might look like this :

```
%module example

%typemap(in) int {
    $1 = $input;
    printf("Received an integer : %d\n", $1);
}

extern int fact(int n);
```

Typemaps require a method name, datatype, and conversion code. The "in" method in this example refers to an input argument of a function. The datatype 'int' tells SWIG that we are remapping integers. The supplied code is used to convert from a jint to the corresponding C datatype, int. Within the supporting C code, the variable \$input contains the Java data (the JNI jint in this case) and \$1 contains the destination of a conversion.

When this example is compiled into a Java module, it can be used as follows:

SWIG and Java

```
System.out.println(example.fact(6));
```

and the output will be:

```
Received an integer : 6
720
```

A full discussion of typemaps can be found in the [SWIG users reference section on typemaps](#). We will primarily be concerned with Java typemaps here.

Java typemaps

The typemaps available to the Java module include the common typemaps listed in the main typemaps section as well as the following Java specific typemaps:

Typemap	Description
typemap(jni)	JNI types. These provide the default mapping of types from C to JNI.
typemap(jtype)	Java types. These provide the default mapping of types from C to Java when used in the Java module class.
typemap(jstype)	Java shadow class types. These provide the default mapping of types from C to Java when used in Java shadow classes.

The default code generated by SWIG for the Java module comes from the typemaps in the java.swg library file. There are other typemaps in the Java library which could be used. These are listed below:

C Type	Typemap	File	Kind	Java Type	Function
char *	BYTE	various.i	input output	byte[]	Java byte array is converted to char array which is released afterwards
char **	STRING_IN	various.i	input	String[]	\0 terminated array of \0 terminated strings the array is malloc-ed and released afterwards
	STRING_OUT	various.i	output	String[]	&char* the argument is the address of an '\0' terminated string
	STRING_RET	various.i	return	String[]	\0 terminated array of \0 terminated strings the array is not free-ed.
primitive type pointers	INPUT	typemaps.i	input	Java basic types	Allows values to be used for c functions taking pointers for data input.
primitive type pointers	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 type pointers	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]	stl_string.i	input output	String	Use for STL std::string mapping to Java String.

If a %typemap(in) is written, a %typemap(freearg) and %typemap(argout) may also need to be written. This is because 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.

Typemap 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

Only in jstype typemaps. \$javaclassname is similar to \$1_basetype, except when applied to unions/structs/classes. Here the type name is used if it has been wrapped by SWIG otherwise a Java long is used. Used primarily to get the Java classname of a wrapped union/struct/class. For example, \$javaclassname is replaced by 'Foo' when the type is 'struct Foo'.

\$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:

```
void jni_fn(...) {
    if (error) {
        SWIG_exception(SWIG_IndexError, "Array element error");
        return ;
    }
    ...
}
```

otherwise \$null expands to NULL

```
jobject jni_fn(...) {
    if (error) {
        SWIG_exception(SWIG_IndexError, "Array element error");
        return NULL;
    }
    ...
}
```

Typemaps for C and C++

JNI calls must be written differently depending on whether the code is being compiled as C or C++. For example C compilation requires syntax like

```
const jclass clazz = (*jenv)->FindClass(jenv, "java/lang/String");
```

whereas C++ code compilation of the same function call has to be written like this

```
const jclass clazz = jenv->FindClass("java/lang/String");
```

To enable typemaps to be compiled as either C or C++ use the JCALLx macros defined in Lib/java/javahead.swg where x is the number of arguments in the C++ version of the JNI call. The above JNI call would be written in a typemap like this

```
const jclass clazz = JCALL1(FindClass, jenv, "java/lang/String");
```

Name based type conversion

Typemaps are based both on the datatype and an optional name attached to a datatype. For example :

```
%module foo

// This typemap will be applied to all char ** function arguments
%typemap(in) char ** { ... }

// This typemap is applied only to char ** arguments named `argv'
%typemap(in) char **argv { ... }
```

In this example, two typemaps are applied to the `char **` datatype. However, the second typemap will only be applied to arguments named ``argv'`. A named typemap will always override an unnamed typemap.

Due to the name-based nature of typemaps, it is important to note that typemaps are independent of typedef declarations. For example :

```
%typemap(in) double {
    ... get a double ...
}
void foo(double);           // Uses the above typemap
typedef double Real;
void bar(Real);             // Does not use the above typemap (double != Real)
```

To get around this problem, the `%apply` directive can be used as follows :

```
%typemap(in) double {
    ... get a double ...
}
void foo(double);

typedef double Real;        // Uses typemap
%apply double { Real };    // Applies all "double" typemaps to Real.
void bar(Real);            // Now uses the same typemap.
```

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 */
%typemap(in) char** (jint size) {
    int 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** {
    int i;
    for (i=0; i<size-1; i++)
        free($1[i]);
    free($1);
}

/* This allows a C function to return a char ** as a Java String array */
%typemap(out) char** {
    int i;
    int len=0;
    jstring temp_string;
    const 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);
    }
}

/* This tells SWIG what the matching JNI type for the Java type is */
%typemap(jni) char** "jobjectArray"

/* This tells SWIG what Java type to use */
%typemap(jtype) char** "String[]"
%typemap(jstype) char** "String[]"

/* 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
}

%}

```

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 now be used by the following Java program:

```

// File main.java
import example;

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

```

Our type-mapping makes the Java interface to these functions more natural and easy to use.

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. 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 pointers 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.

```
/* 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 and use some typemaps we can solve this problem, for example we could put the following through SWIG:

```
%pragma make_default
%module example

%{
/* 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;
};
%}

/* Define a new structure to use instead of double* */
%inline %{
typedef struct {
    double value;
} MyDouble;
%}

/*
This typemap will make any double* function parameters with name 'OutValue' take an
argument of MyDouble instead of double*. Requires -shadow commandline option. 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, "_cPtr", "J");
    jlong cPtr = jenv->GetLongField($input, fid);
    MyDouble* pMyDouble;
    *(MyDouble*)= *(MyDouble*)
    $1 = value;
}

/* This tells SWIG what Java type to use */
%typemap(jtype) double* OutValue "MyDouble"
%typemap(jstype) double* OutValue "MyDouble"

/* This tells SWIG what the matching JNI type for the Java type is */
%typemap(jni) double* OutValue "jobject"
```

```
/* 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. Also this will only work with shadow classes enabled (`-shadow` commandline option) as the `MyDouble` shadow class contains the member variable `'_cPtr'` which we need in the `'in'` typemap. 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, is then able to get the `'double value'` member variable to pass 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
import example.*;

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
```

Accessing array structure members

Consider the following data structure :

```
#define LEN 5
typedef struct {
    int numbers[LEN];
} Data;
```

By default, the Java module supplies the `memberin` typemap in the `Lib/java/java_arrays` file to set array members. It is generic, but not particularly efficient as it copies one array member at a time. You may want to override it for the above `Data` struct as follows:

```
%typemap(memberin) int[LEN] {
```

```

/* Copy at most LEN characters into $1 */
memcpy($1,$input,sizeof(int)*LEN);
}

```

Whenever a `int[LEN]` type is encountered in a structure or class, this typemap provides a safe mechanism for setting its value. An alternative implementation might choose to print an error message if the Java supplied array was too long to fit into the field.

It should be noted that the `[LEN]` array size is attached to the typemap. A datatype involving some other kind of array would not be affected. However, you can write a typemap to match any sized array using the `ANY` keyword as follows :

```

%typemap(memberin) int [ANY] {
    memcpy($1,$input,sizeof(int)*$dim0);
}

```

During code generation, `$dim0` will be filled in with the real array dimension.

Pointer handling

Unlike other language modules, the Java SWIG pointer library has not been written. The consequence of this is that there is no access to the SWIG runtime type checker which is used by the pointer library. This also means that your own pointer handling functions will have to be written or use some of the concepts shown in the typemap examples. The `typemaps.i` library file will be useful for pointer handling as previously mentioned. All is not lost as this does not apply when using shadow classes and non-primitive data types, as the section on Shadow classes demonstrates.

Pointers are stored in a Java long, which is a 64 bit number. However most JVMs are 32 bit applications so any JNI code must also be compiled as 32 bit. This means that the pointers in JNI code are also 32 bits. What happens for various reasons is on big endian machines the pointer is stored in the high order 4bytes, whereas on little endian machines the pointer is stored in the low order 4bytes. As a result, care must be taken if you intend to manipulate the pointer directly from Java. This can of course be changed with judicious use of typemaps, but normally you needn't worry about any of this unless you want to modify pointers within Java code.

By now you hopefully have the idea that typemaps are a powerful mechanism for building more specialized applications. While writing typemaps can be technical, many have already been written for you. See the Typemaps chapter for more information about using library files.

The gory details of shadow classes

This section describes the process by which SWIG creates shadow classes and some of the more subtle aspects of using them.

A simple shadow class

Consider the following barebones C++ class:

```

%module example

```

```
%inline %{

class Simple {
public:
    Simple() {};
    ~Simple() {};
};

%}
```

The SWIG generated class in file Simple.java looks like the following:

```
// File Simple.java
import example;

public class Simple {
    protected long _cPtr;
    protected boolean _cMemOwn;

    public Simple(long cPointer, boolean cMemoryOwn) {
        _cPtr = cPointer;
        _cMemOwn = cMemoryOwn;
    }

    public long getCPtr() {
        return _cPtr;
    }

    public Simple() {
        _cPtr = example.new_Simple();
        _cMemOwn = true;
    }

    protected void finalize() {
        _delete();
    }

    public void _delete() {
        if(_cPtr!=0 & _cMemOwn) {
            example.delete_Simple(_cPtr);
            _cPtr = 0;
        }
    }
}
```

Generated class

Shadow classes are built using the low-level SWIG generated C interface. The simple interface is wrapped into a Java class named after the module name, 'example' here. The Java code for the shadow class is created in many different files. The name of each file is the name of the shadow class, which in turn is named after the class/struct/union that is being shadowed (proxied). This has to be the case as in Java a class called 'A' must be in a file called A.java. Except for a couple of memory management methods and constructors, the methods in the shadow class simply call the SWIG generated functions in the simple interface. The shadow class is a lot easier to use as it wraps the C++ 'this' pointer making memory management a lot easier and less error prone. Shadow classes generally fit in with the Java programming paradigm.

The this pointer

Each generated shadow class has the above member variables and functions, except derived classes, which will be covered later. The `_cPtr` holds the pointer to an instance of the C/C++ class/struct/union, so it contains the C++ 'this' pointer. If you need the pointer it can be obtained by using the `getCPtr()`, as `_cPtr` is protected data. The shadow classes have been written with easy memory management in mind for most cases so you should normally not need to use `_cPtr`, nor its public accessor function `getCPtr()`. The `Grid2d` class, on the other hand, is used when you want to create a new `Grid2d` object from Java. In reality, it inherits all of the attributes of a `Grid2dPtr`, except that its constructor calls the corresponding C++ constructor to create a new object. Thus, in Java, this would look something like the following :

```
>>> g = Grid2d(50,50)           # Create a new Grid2d
>>> g.xpoints
50
>>>
```

Object ownership

Ownership is a critical issue when mixing C++ and Java. For example, suppose I create a new object in C++, but later use it to create a Java object. If that object is being used elsewhere in the C++ code, we clearly don't want Java to delete the C++ object when the Java object is deleted. Similarly, what if I create a new object in Java, but C++ saves a pointer to it and starts using it repeatedly. Clearly, we need some notion of who owns what. Since sorting out all of the possibilities is probably impossible, the shadow class always has an attribute "`_cMemOwn`" that indicates whether or not Java owns an object.

The default object ownership falls into 2 cases:

1. Whenever an object is created in Java, Java will be given ownership by setting `_cMemOwn` to `true`.
2. When a Java class is created from a pre-existing SWIG generated shadow class, ownership is assumed to belong to the C/C++ code and `_cMemOwn` will be set to `false`.

When `_cMemOwn` is set, Java will attempt to call the C/C++ destructor when the object is deleted, that is garbage collected. If it is zero, Java will never call the C/C++ destructor. Ownership of an object is set up using the appropriate constructors. Note that sometimes the garbage collector does not call finalizers. Please see the section on Shadow classes and Garbage collection for more information.

Constructors and destructors

C++ constructors are mapped into an equivalent Java constructor. An additional constructor is available for use which overrides the default memory handling/memory ownership. For the above example, this constructor is shown as:

```
public Simple(long cPointer, boolean cMemoryOwn) {
    _cPtr = cPointer;
    _cMemOwn = cMemoryOwn;
}
```

It allows one to override the above default `_cPtr` and `_cMemOwn` values on construction.

The `finalize()` method is generated by default, but it can be turned off using the `-nofinalize` commandline option. It calls the `_delete()` method which calls the C++ destructor only if the Java object owns the C++ memory. If there is no C++ default constructor, a default constructor is required by Java and so a protected constructor is generated as follows:

```
protected Simple() {
    _cPtr = 0;
    _cMemOwn = false;
}
```

It is highly recommended to use the `%make_default` directive in your interface file so that SWIG generates default constructors and destructors if none exist. This is useful as these do not exist if you are using C. Note that when using C++ the compiler generates a default constructor and destructor for you if none exist, so this is different to SWIG's default behaviour. Note that SWIG currently does not support method overloading and thus constructor overloading, but this is likely to change in a future release.

Member data

Member data of a C/C++ object is accessed through getter and setter methods. For example, if a public member variable and a public static member variable is added to the previous bare bones class as such:

```
class Simple {
public:
    Simple() {} ;
    ~Simple() {} ;
    int Number;
    static int StaticNumber;
};
```

The class can be used as follows to read and write to the variables:

```
Simple simp = new Simple();
simp.setNumber(10);
int number = simp.getNumber();

Simple.setStaticNumber(10);
number = Simple.getStaticNumber();
```

Note that the C++ static member variable is mapped using Java static getters and setters.

Shadow Functions

Suppose you have the following declarations in an interface file :

```
%module example
struct Vector {
    Vector();
    ~Vector();
    double x,y,z;
    static Vector addv(Vector a, Vector b);
};
```

By default in the simple SWIG interface, the function `addv` will operate on Vector pointers (a Java long), not Java classes:


```
// Member function in class example (SWIG's simple interface)
public final static native long Vector_addv(long jarg0, long jarg1);
```

However, when using shadow classes, the Java SWIG module is smart enough to know that `Vector` has been wrapped into a Java class so it will create the following shadow function for the `addv()` function.

```
// Member function in class Vector (the shadow class)
public static Vector addv(Vector a, Vector b) {
    return new Vector(example.Vector_addv(a.getCPtr(), b.getCPtr()), true);
}
```

Function arguments are modified to pick up the "this" pointer from a Java `Vector` object. The return value is a pointer to the result which has been allocated by `malloc` or `new` (this behavior is described in the chapter on SWIG basics), so we simply create a new `Vector` class with the return value. Since the result involved an implicit `malloc`, we set the ownership to `true` indicating that the result is to be owned by Java and that it should be deleted when the Java object is deleted. As a result, operations like this are perfectly legal and result in no memory leaks:

```
Vector v = Vector.addv(Vector.addv(Vector.addv(Vector.addv(a,b),c),d),e);
```

Shadow class pointer handling

Now let's take the previous example and in a new function `cross_product` and use `Vector` pointers for the parameters as well as the return type:

```
%module example
struct Vector {
    Vector();
    ~Vector();
    double x,y,z;
    static Vector addv(Vector a, Vector b);
    static Vector *cross_product(Vector *v1, Vector *v2);
};
```

By default in the simple SWIG interface, the function `cross_product` will also operate on `Vector` pointers (a Java long):

```
// Member function in class example (SWIG's simple interface)
public final static native long Vector_cross_product(long jarg0, long jarg1);
```

When using shadow classes, the Java SWIG module still allows us to use a `Vector` proper:

```
// Member function in class Vector (the shadow class)
public static Vector cross_product(Vector v1, Vector v2) {
    return new Vector(example.Vector_cross_product(v1.getCPtr(), v2.getCPtr()), false);
}
```

The shadow function gets the pointer for us from the `Vector` class. Note that the return value is a `Vector` whose memory ownership is set to `false` indicating that the memory is not owned by Java. The following section discusses this further.

An important observation to note when using shadow classes is the differences in wrapping pointers to complex

datatypes and pointers to primitive types. Pointers to complex data types are wrapped using a Java class, such as `Vector` for `Vector*`. Pointers to primitive types are wrapped in the same manner as the simple interface, that is a Java `long` for `short*`, `int*` etc.

Methods that return new objects

By default SWIG assumes that constructors are the only functions returning new objects to Java. However, you may have other functions that return new objects as well. If we take our previous `Vector` example and examine the `cross_product` implementation:

```
Vector *Vector::cross_product(Vector *v1, Vector *v2) {
    Vector *result = new Vector();
    result = ... compute cross product ...
    return result;
}
```

When the value is returned to Java, we want Java to assume ownership as the memory has been created on the heap. However, SWIG has no way of knowing that it should take ownership, so by default it does not. The memory has to be deleted after calling the `cross_product` function:

```
Vector a = new Vector();
Vector b = new Vector();
// ... populate a and b ...

Vector c = Vector.cross_product(a, b);

// clean up the memory allocated by cross_product
example.delete_Vector(c.getCPtr());
```

Unfortunately, this is ugly, is likely to be forgotten and it doesn't work if we use the result as a temporary value :

```
Vector w = Vector.addv(Vector.cross_product(a,b),c);    // Results in a memory leak
```

However, you can provide a hint to SWIG when working with such a function as shown :

```
// C/C++ Function returning a new object
struct Vector {
    ...
    %new static Vector *cross_product(Vector *v1, Vector *v2);
    ...
};
```

The `%new` directive provides a hint that the function is returning a new object. The Java module will assign proper ownership of the object when this is used. This can be seen as this time it uses `true` in the constructor:

```
// Member function in class Vector (the shadow class)
public static Vector cross_product(Vector v1, Vector v2) {
    return new Vector(example.Vector_cross_product(v1.getCPtr(), v2.getCPtr()), true);
}
```

Global variables and functions

Substitution of complex datatypes occurs for all shadow functions and shadow member functions involving structures, unions or class definitions. Currently no shadow classes are produced for global variables nor global functions. Only the low-level C interface is produced. This is an important limitation. The recommended approach is to put all global access into a dummy structure with static function access:

```
%module example
%pragma make_default

%inline %{
struct Vector {
    Vector() {} ;
    ~Vector() {} ;
    double x,y,z;
    static Vector addv(Vector a, Vector b);
};

// Global function and variable
Vector global_subv(Vector a, Vector b);
Vector global_vec;

// Global function and variable wrapped into this structure
struct Globals {
    static Vector subl(Vector a, Vector b) { return global_subv(a, b); };
    static void setGlobal_vec(Vector v) { global_vec = v; };
    static Vector getGlobal_vec(void) { return global_vec; };
};
%}

// Alternative way to wrap global functions into a structure
%addmethods Globals { static Vector sub2(Vector a, Vector b) { return global_subv(a, b); } }
```

The dummy class consists entirely of static member functions. Note the two different ways that `global_subv` can be wrapped into `Globals`. Access to the global function `global_subv` is then much neater:

```
Vector a = new Vector();
Vector b = new Vector();
// ... setup a and b ...

// Access to global function which has been wrapped into the Globals shadow class
Vector c1 = Globals.sub1(a,b);
Vector c2 = Globals.sub2(a,b);

// Access to global function using simple interface
Vector c3 = new Vector(example.global_subv(a.getCPtr(), b.getCPtr()), true);
```

This is one area where the Java module needs improving in the future so that all global functions are for instance automatically accessible through a shadow class.

Nested objects

SWIG shadow classes support nesting of complex objects. For example, suppose you had the following interface file :

```
%module example
%pragma make_default

%inline %{

struct Vector {
    double x,y,z;
    static Vector addv(Vector a, Vector b);
};

typedef struct {
    Vector r;
    Vector v;
    Vector f;
    int    type;
} Particle;

%}
```

In this case you will be able to read and write to members as follows :

```
Vector v1 = new Vector();
Vector v2 = new Vector();
// ... populate v1 and v2 ...

Particle p = new Particle();
p.getR().setX(0.0);
p.getR().setY(-1.5);
p.getR().setZ(2.0);
p.setV( Vector.addv(v1, v2) );
```

Nested structures such as the following are also supported by SWIG. These types of structures tend to arise frequently in database and information processing applications.

```
typedef struct {
    unsigned int dataType;
    union {
        int      intval;
        double   doubleval;
        char     *charval;
        void     *ptrvalue;
        long     longval;
        struct {
            int    i;
            double f;
            void   *v;
            char   name[32];
        } v;
    } u;
} ValueStruct;
```

Access is provided like this:

```
ValueStruct v = new ValueStruct();
// ... populate ValueStruct somehow ...

long dataType = v.getDataType();
int intval = v.getU().getIntval();
```

```
double f = v.getU().getV().getF();
```

To support the embedded structure definitions, SWIG has to extract the internal structure definitions and use them to create new Java classes. In this example, the following shadow classes are created:

```
//Class corresponding to union u member
ValueStruct_u u = v.getU();

//Class corresponding to struct v member of union u
ValueStruct_u_v uv = v.getU().getV();
```

The names of the new classes are formed by appending the member names of each embedded structure.

Inheritance and shadow classes

Since shadow classes are implemented in Java, you can use any of the automatically generated classes as a base class for more Java classes.

SWIG can detect C++ classes that are abstract. This means that the Java shadow class can accurately shadow the C++ class. For example, given the abstract base class Shape and its derived class Circle:

```
class Shape {
    double  x, y;
public:
    Shape();
    virtual ~Shape();
    void    move(double dx, double dy);
    virtual double area() = 0;
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) { };
    virtual double area();
};
```

The Shape shadow class becomes an abstract Java class with the pure virtual function area being declared as a Java abstract method. The constructor is also made protected:

```
// File Shape.java
import example;

public abstract class Shape {
    protected long _cPtr;
    protected boolean _cMemOwn;

    public Shape(long cPointer, boolean cMemoryOwn) {
        _cPtr = cPointer;
        _cMemOwn = cMemoryOwn;
    }

    public long getCPtr() {
        return _cPtr;
    }
}
```

```

protected Shape() {
    _cPtr = 0;
    _cMemOwn = false;
}

protected void finalize() {
    _delete();
}

public void _delete() {
    if(_cPtr!=0 & _cMemOwn) {
        example.delete_Shape(_cPtr);
        _cPtr = 0;
    }
}

public void move(double dx, double dy) {
    example.Shape_move(_cPtr, dx, dy);
}

public abstract double area();
}

```

The Circle class extends Shape mirroring the C++ class inheritance hierarchy.

```

// File Circle.java
import example;

public class Circle extends Shape {
    public Circle(long cPointer, boolean cMemoryOwn) {
        super(cPointer, cMemoryOwn);
    }

    protected Circle() {
    }

    public Circle(double r) {
        super(0, false);
        _cPtr = example.new_Circle(r);
        _cMemOwn = true;
        // The above will be optimised in the future to:
        // super(example.new_Circle(r), true);
    }

    public double area() {
        return example.Circle_area(_cPtr);
    }
}

```

It is then of course possible to extend Shape using your own Java classes. If say Circle is provided by the C++ code, you could for example add in a pure Java class Rectangle. There is a caveat and that is any C++ code will not know about your Java class Rectangle so this type of derivation is restricted.

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.

Shadow classes and garbage collection

If you get SWIG to produce shadow classes, you will notice the generated `_delete()` and `finalize()` methods. The `finalize()` method calls `_delete()` which frees any SWIG malloced c memory for wrapped structs or deletes any SWIG wrapped classes created on the heap, which in turn calls the class' destructor. 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 Windows and Unix. Say your JNI code creates memory on the heap which your finalizers will 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, 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 the garbage collector will eventually run, so long running programs will have their unreferenced object's finalizers called.

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. 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. Currently you cannot tell SWIG to derive a SWIG shadow class from any named Java class, but this is planned in the near future. An alternative is to add the object handling code to all generated shadow classes using the shadow pragma.

Performance concerns and hints

Shadow classing is primarily intended to be a convenient way of accessing C/C++ objects from Java. However, if you're directly manipulating huge arrays of complex objects from Java, performance may suffer greatly. In these cases, you should consider implementing the functions in C or thinking of ways to optimize the problem. 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.

If performance is really critical you can use the low-level interface which eliminates all of the overhead of going through the shadow classes (at the expense of coding simplicity).

Java pragmas

There are two groups of pragmas in the Java module. The first group modify the Java module output file and the second modify the Java shadow output file. The pragmas beginning with **module** apply to the module output file. The pragmas beginning with **allshadow** apply to all shadow output classes. The pragmas beginning with **shadow** apply to the currently active shadow class so these pragmas can only be specified within the definition of a class or struct. Some examples:

```
/* TestStruct.i */
#ifdef SWIG
#pragma(java) modulecode=%{ /*This code gets added to the module class*/ %}
#pragma(java) modulecode=%{ /*This code gets added to the module class*/ %}
#pragma(java) allshadowcode=%{ /*This code gets added to every shadow class*/ %}
#endif
typedef struct TestStruct {
#ifdef SWIG
#pragma(java) shadowcode=%{ /*This code gets added to the TestStruct shadow class only*/ %}
#endif
    int    myInt;
} TestStruct;
```

Note that pragmas will take either " " or %{ %} as delimiters. If the pragma requires the string quote it must be preceded with a backslash or alternatively use the %{ %} delimiters, for example either of these can be used to

obtain the desired effect:

```
%pragma(java) modulecode= "    public void sayHello() { System.out.println(\"Hello\"); } "
%pragma(java) modulecode=%{    public void sayHello() { System.out.println("Hello"); } %}
```

A complete list of pragmas follows:

Pragma	Description	Example
modulebase	Specifies a base class for the Java module class.	%pragma(java) modulebase="BaseClass"
shadowbase	Specifies a base class for the Java shadow class.	%pragma(java) shadowbase="BaseClass"
allshadowbase	Specifies a base class for all Java shadow classes.	%pragma(java) allshadowbase="BaseClass"
modulecode	Adds code to the Java module class.	%pragma(java) modulecode=% { /*module code*/% }
shadowcode	Adds code to the Java shadow class. See the JavaDoc section below about using this pragma for JavaDoc comments.	%pragma(java) shadowcode=% { /*shadow code*/% }
allshadowcode	Adds code to all Java classes.	%pragma(java) allshadowcode=% { /*all shadow code*/% }
moduleclassmodifiers	Overrides the default Java module class modifiers. The default is public.	%pragma(java) moduleclassmodifiers="public final"
shadowclassmodifiers	Overrides the default Java shadow class modifiers. The default is public. Also overrides allshadowclassmodifiers if present.	%pragma(java) shadowclassmodifiers="public final"
allshadowclassmodifiers	Overrides the default modifiers for all Java classes. The default is public.	%pragma(java) allshadowclassmodifiers="public final"
moduleimport	Adds an import statement to the Java module class file.	%pragma(java) moduleimport="java.lang.*"
shadowimport	Adds an import statement to the Java shadow class file. Adds to any imports specified in allshadowimport.	%pragma(java) shadowimport="java.lang.*"
allshadowimport	Adds an import statement to all Java shadow class files.	%pragma(java) allshadowimport="java.lang.*"
moduleinterface	Specifies an interface which the Java module output class implements. Can be used multiple times as Java supports multiple interfaces.	%pragma(java) moduleinterface="SomeInterface"
shadowinterface		%pragma(java) shadowinterface="SomeInterface"

	Specifies an interface which the Java shadow class implements. Can be used multiple times as Java supports multiple interfaces. Adds to any interfaces specified in <code>allshadowinterface</code> .	
<code>allshadowinterface</code>	Specifies an interface which all Java shadow classes implement. Can be used multiple times as Java supports multiple interfaces.	<code>%pragma(java) allshadowinterface="SomeInterface"</code>
<code>modulemethodmodifiers</code>	Overrides the native default method modifiers for the module class. The default is public final static.	<code>%pragma(java) modulemethodmodifiers="protected final static synchronized"</code>

Deprecated pragmas

The following pragmas were in Swig 1.3a3 and have since been deprecated:

import: Please replace with **moduleimport**, **shadowimport** and/or **allshadowimport** pragmas.

module: Please replace with the **modulecode** pragma.

shadow: Please replace with the **allshadowcode** pragma.

modifiers: Please replace with the **modulemethodmodifiers** pragma.

Pragma uses

The pragmas are used primarily to modify the default Java output code. They provide flexibility as to how Java and C++ interact. In the pragma notation below, **xxxcode** for example would be used to denote the 3 similar 'code' pragmas, that is **modulecode**, **shadowcode** and **allshadowcode**.

Derive C++ from Java and visa-versa

It is possible to derive a Java class from a shadow class (i.e. a C++ class) with the Java module. However, with the pragmas it is also possible to derive the SWIG produced Java shadow classes (i.e. C++ class) from your own Java class by using the **xxxbase** pragma with the **xxximport**, **xxxcode** and **xxxclassmodifiers** pragmas. It is also possible for the SWIG produced Java classes to implement interfaces using the **xxxinterface** pragmas.

JavaDoc and the shadowcode pragma

The SWIG documentation feature is not currently present in the 1.3 versions of SWIG. However a limited form of JavaDoc documentation can be implemented by placing JavaDoc comments in the **shadowcode** pragma. Just put your JavaDoc comment in a **shadowcode** pragma before the function to which you want it to apply.

Tips for using the shadow pragmas

Note that an out of scope warning is issued if any of the **shadow** pragmas are used outside of the class/struct/union definition. This requires one of these pragmas to be placed within the definition of the class/struct/union. It may seem that the original C/C++ code has to be modified when using the `%include` pragma. A technique to avoid this is to place the pragma within the `%addmethods` directive. For example:

```
%module example
%{
```

```

#include "MyClass.h"
%}
#include "MyClass.h"

// Use a shadow pragma without modifying the original code in MyClass.h
%addmethods MyClass{
%pragma(java) shadowcode="/* Some extra shadow code for MyClass */"
}

```

Dynamic linking problems

The code to load a native library is `System.loadLibrary("name")`. This can fail and it can be due to a number of reasons.

The most common is an incorrect naming of the native library for the name passed to the `loadLibrary` function. The text passed to the `loadLibrary` function must not include the extension name in the text, that is `.dll` or `.so`. The text must be *name* and not *libname* for all platforms. On Windows the native library must then be called *name.dll* and on Unix 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 Unix make sure that your `LD_LIBRARY_PATH` contains the path to the native library. On Windows make sure the *path* environment variable contains the path to the native library. SWIG code usually works if you have `LD_LIBRARY_PATH` set to `'.'` (or no modification to *path* in Windows).

The native library will also not load if there are any unresolved symbols in the compiled C/C++ code. Unresolved symbols will be described if you use the `-verbose:jni` commandline switch when running java.

Ensure that you are using the correct C/C++ compiler and linker combination and options for successful native library loading. 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.

Linking problems used to occur when there were underscores in the module name or package name. This was fixed in SWIG 1.3.7.

Tips

The `%native` directive can be used to mix hand written JNI functions with the auto generated functions.

Known bugs

- Function pointers produce code that won't compile. Your own typemaps will overcome this.

G SWIG and Guile

- [G.1 Meaning of "Module"](#)
- [G.2 Linkage](#)
- [G.3 Underscore Folding](#)
- [G.4 Typemaps](#)
- [G.5 Smobs](#)
- [G.6 Exception Handling](#)
- [G.7 Procedure documentation](#)
- [G.8 Procedures with setters](#)

This section details guile-specific support in SWIG.

G.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".

G.2 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**.

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 foo.scm`. 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 *foo.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.

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.

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")
```

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.

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.

General Remarks on Multiple SWIG Modules

If you want to use multiple SWIG modules, they have to share some run-time data for the typing system. You have two options:

- Either generate all but one wrapper module with the `-c` command-line argument. Compile all wrapper files with the C compiler switch `-DSWIG_GLOBAL`.
- Or generate all wrapper modules with the `-c` command-line argument and compile all wrapper files with the C compiler switch `-DSWIG_GLOBAL`. Then link against the runtime library `libswigguile`, which is built by `make runtime`. The needed linker flags are reported by SWIG if you invoke it with the `-guile -ldflags` command-line arguments.

[G.3 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).

G.4 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:

- `$descriptor` expands to a type descriptor for use with the `SWIG_Guile_MakePtr()` and `SWIG_Guile_GetPtr` functions.
- For pointer types, `*$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 `GH_UNSPECIFIED`) is treated as returning no values. In `argout` typemaps, one can use the macro `GUILE_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.

G.5 Smobs

For pointer types, SWIG uses Guile 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. 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_Guile_MakePtr()`, 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_Guile_GetPtr`, passing a pointer to a struct representing the expected pointer type. If the Scheme object passed was not a SWIG smob representing a compatible pointer, a `wrong-type-arg` exception is raised.

G.6 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.

G.7 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 typemaps `indoc`, `outdoc`, `argoutdoc`, `varindoc`, and `varoutdoc`. See `Lib/guile/typemaps.i` for details.

G.8 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`, 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.

SWIG and Ruby

- [Preliminaries](#)
- [Building Ruby Extensions under Windows 95/NT](#)

This chapter describes SWIG's support of Ruby.

Note that this chapter is in its early infant stage and only really has some advice for using SWIG and Ruby on Windows.

Preliminaries

SWIG 1.3 is known to work with Ruby 1.6.4 and Ruby 1.6.5, but should work with other versions. Given the choice, you should use the latest 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.

Running SWIG

As described in the [introduction](#), a module is created from an interface file, which contains function prototypes and variable declarations. For `example.c` this would conventionally be called `example.i`. Often it may be possible to use the C file itself as if it were a `.i` file, as described in the Shortcuts section of the [introduction](#).

To build a Ruby module, run SWIG using the `-ruby` option :

```
%swig -ruby example.i
```

It should be noted that if the `.c` file is used instead of the `.i` file, then the `-module Example` must be used on the command line:

```
%swig -ruby -module Example example.i
```

Alternatively, a real `.i` file may be created containing

```
%module Example
```

Swig will create `example_wrap.c` which together with `example.c` can then be compiled and linked to produce `example.so`. This library can then be handled by Ruby's `require` statement.

If many `.c` files are to be used together in the same module, then several `.i` files may be tied together by creating one overall `.i` file

```
%module MyModule
#include somefuncs.i
#include someotherfuncs.i
```

Compiling a dynamic module

Conventionally with SWIG on Unix the compilation of examples is done using the file `Example/Makefile`. This makefile performs a manual module compilation which is platform specific. Typically, the steps look like this (Linux):

```
% swig -ruby example.i
% gcc -fpic -c example_wrap.c -I/usr/local/lib/ruby/1.4/i686-linux
% gcc -shared example_wrap.o $(OBJS) -o example.so
```

However the politically "correct" way to compile a Ruby extension is to follow the steps described `README.EXT` in Ruby distribution:

1. Create a file called `extconf.rb` that looks like the following:

```
require 'mkmf'
create_makefile('interface')
```

2. Type the following to build the extension:

```
% ruby extconf.rb
% make
% make install
```

Because the compilation step is performed here, there should be no need to invoke the SWIG `Examples/Makefile`

Using your module

The library produced by SWIG contains a module in the Ruby sense. It may be accessed by having statements such as

```
require "Example"

Example.my_function(this, that, other)
```

in the code.

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. This section covers the process of using SWIG with Microsoft Visual C++ 6 although the procedure may be similar with other compilers. In order to build extensions, you will need to download the source distribution to the Ruby package as you will need the Ruby header files.

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 -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 (ie. `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.
- Finally still under the Link tab, add the dll entry point in the Project Options: Use `"/EXPORT:Init_example"` for when you have set the swig module name to 'example'. In general use `"/EXPORT:Init_"`, where is the swig module name (specified using `%module` in your interface file).
- 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
```

SWIG and PHP4

Caution: This chapter (and module!) is still under construction

- [Preliminaries](#)
- [Building PHP4 Extensions](#)
- [Building a loadable extension](#)
- [Basic PHP4 interface](#)
- [Simple Example](#) (TODO)
- [Constructors and Destructors](#)
- [Arrays and other objects](#) (TODO)
- [Remapping datatypes with typemaps](#) (TODO)
- [Constants](#)
- [PHP4 Pragmas](#)
- [Building extensions into PHP](#)

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!.)

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.

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.

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`.)

Before running the 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.

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](#).

Basic PHP4 interface

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
```

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.

Pointers

Pointers to C/C++ objects are represented as character strings such as the following :

```
_523d3f4_Circle_p
```

A NULL pointer is represented by the string "NULL". You can also explicitly create a NULL pointer consisting of the value 0 and a type such as :

```
_0_Circle_p
```

In theory you should never need to see these strings in your PHP script.

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 shadow classes that hides these functions and uses an object oriented interface instead – see [below](#)

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 an 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.)

Shadow 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" an underlying C/C++ class. To have SWIG create shadow classes, use the `-shadow` option :

```
% swig -php4 -shadow tbc.i
```

This will produce the same files as before except that the .php file will now contain significantly more support PHP code. (example on the way) 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)

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.

Destructors do not exist in PHP, as PHP expects to handle all memory allocation and cleaning up for you. This can create problems with wrapped code as memory allocated by the C library may not be freed. However, the function 'register_shutdown_function' can be used to call a function or a object method at the end of a request. In shadow classes this method is used to call a '_destroy' method which then calls the objects destructor to free the memory or perform any other cleanup needed. However, you cannot use print/echo with the shutdown_function as the request has already been completed. It also means that objects are all freed at the end of a script, not when they each go out of scope.

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";
```

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"
```

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.

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.)

SWIG and PHP4

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.

To be furthered...

Extending SWIG

Caution: This chapter is being rewritten! (11/25/01)

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 to find a few inconsistencies.

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, "Kand the "C++ Annotated Reference Manual" by Stroustrup (a.k.a, the "ARM") will be of great use.

High Level Overview

When you run SWIG on an interface, processing is handled in stages by a few different 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. For the most part, this tree includes information about everything that appeared in the input. However, certain features such as C++ templates and `%rename` directives are handled by the parser and may not appear in the resulting parse tree (at least not as special parse tree nodes). It is also important to emphasize that 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.
- One or more code generation components walk the parse tree in order produce wrapper code. All of SWIG's target languages are currently implemented in this stage of processing.

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

Extending SWIG

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 %readonly      %pragma(swig) readonly;
#define %readwrite    %pragma(swig) readwrite;

/* Directives for callback functions */

#define %callback(x) %pragma(swig) callback=`x`;
#define %nocallback  %pragma(swig) nocallback;

/* Directives for attribute functions */

#define %attributefunc(_x,_y) %pragma(swig) attributefunction=`_x`:"`_y`;
#define %noattributefunc    %pragma(swig) noattributefunction;

/* %ignore directive */

#define %ignore          %rename($ignore)
#define %ignorewarn(x) %rename("$ignore:" x)

/* Generation of default constructors/destructors */

#define %nodefault      %pragma nodefault
#define %makedefault    %pragma makedefault

...
```

The fact that most of the standard SWIG directives are macros is intended to simplify the implementation of the parser. For instance, rather than having to support dozens of special grammar rules, it is easier to have a few basic primitives such as `%pragma` 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`.

Although the SWIG preprocessor is intended to mimic the behavior of the C preprocessor, it is not meant to be a direct replacement. Instead, its behavior is adapted for use with SWIG and it provides a number of a non-standard extensions:

- File inclusion directives such as `#include <stdio.h>` are ignored by default.
- When files are included, they are enclosed in a SWIG file-scope block that looks like this:

```
%includefile "example.i" {
...
}
```

These blocks are used during parsing and code generation to track the relationship between all of the files that were included or imported into an interface. This is especially important when SWIG is being used to create a collection of interrelated extension modules.

- Preprocessing is disabled inside any `%{ ... %}` block.
- `#define` statements are examined to see if they might be constants. If so, they are transformed into `%constant` declarations for the parser. A `#define` statement is assumed to be a constant if it does not contain any unresolved identifier names when expanded.
- Macro expansion supports the non-standard ``x`` operator. This converts `x` into a quoted string unless it is already a quoted string (in which case it remains quoted). This is sometimes used to define SWIG directives that support both quoted and non-quoted arguments.
- Macro names can start with `'%'`. This allows SWIG directives to be defined as macros.
- Macros can be defined using `%define` and `%enddef`. For example:

```
%define FOO(a,b)
...
%enddef
```

These macros differ from C preprocessor macros in two respects. First, they can span multiple lines. Second, when the macros are expanded, the expanded text is re-parsed by the preprocessor.

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.

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 pieces. For example, it is technically legal, although unusual to write something

Extending SWIG

like `int typedef const a` in your program. SWIG simply doesn't bother to deal with this (although it could probably be modified if there is sufficient demand).

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. On a more practical level however, the introduction of typenames would greatly complicate other parts of SWIG such as the parsing of SWIG directives (many of which also rely upon identifier names).

Because of the different treatment of typenames, the most serious limitation of the SWIG parser is that it can't process type declarations in which 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).

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`. For example:

```
$ swig -c++ -python -dump_tags example.i
. top (example.i:1)
. top . include (example.i:1)
. top . include . typemap (/r0/beazley/Projects/lib/swig1.3/swig.swg:71)
. top . include . typemap . typemapitem (/r0/beazley/Projects/lib/swig1.3/swig.swg:71)
. top . include . typemap (/r0/beazley/Projects/lib/swig1.3/swig.swg:83)
```

[illegible]

[illegible]

```

. top . include . typemap . typemapitem (/r0/beazley/Projects/lib/swig1.3/python/python.swg
. 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 . 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 stored as a hash table. 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"
| 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). "
| parms         - double ,double
| type          - "void"
| sym:symtab    - 0x40194140
|
+++ cdecl -----
| sym:name      - "area"
| name          - "area"
| decl          - "f(void). "
| parms         - void
| storage       - "virtual"
| value         - "0"
| type          - "double"
| sym:symtab    - 0x40194140
|
+++ cdecl -----
| sym:name      - "perimeter"
| name          - "perimeter"
| decl          - "f(void). "
| parms         - void
| storage       - "virtual"
| value         - "0"
| type          - "double"
| sym:symtab    - 0x40194140
|
```

```

+++ cdecl -----
| sym:name      - "nshapes"
| name          - "nshapes"
| decl          - ""
| storage       - "static"
| type          - "int"
| 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

```

Attribute namespaces

When attributes are added to parse tree nodes, their names may be 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 very general attributes such as types, names, and so forth appear without a prefix.

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`.

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 )
```

The %feature directive

A number of SWIG directives such as `%exception` are implemented using the lower-level `%feature` directive. For example:

```
%feature("except") getitem(int) {
    try {
```

Extending SWIG

```
        $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 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 name that it wishes. 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.

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 void import_start(char *);
    virtual void import_end();
};
```

The role of these functions are described shortly.

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.

Summary so far

SWIG is a multi-pass compiler that works by building a complete parse tree of input files. These parse trees are structured as a hierarchy of nodes with arbitrary attributes. Language modules are created by writing special handlers for different types of parse tree nodes.

The rest of this chapter describes some of the internal data structures and various code generation tasks in more detail.

SWIG 1.3 – Last Modified : January 22, 2002

11 Advanced Topics

Caution: This chapter is under repair!

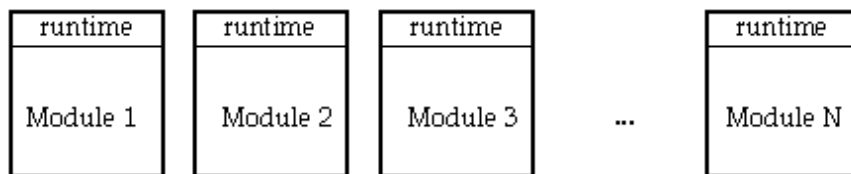
- [Creating multi-module packages](#)
- [Dynamic Loading of C++ modules](#)
- [Inside the SWIG type-checker](#)

Creating multi-module packages

SWIG can be used to create packages consisting of many different modules. However, there are some technical aspects of doing this and techniques for managing the problem.

Runtime support (and potential problems)

All SWIG generated modules rely upon a small collection of functions that are used during run-time. These functions are primarily used for pointer type-checking, exception handling, and so on. When you run SWIG, these functions are included in the wrapper file (and declared as static). If you create a system consisting of many modules, each one will have an identical copy of these runtime libraries :



This duplication of runtime libraries is usually harmless since there are no namespace conflicts and memory overhead is minimal. However, there is serious problem related to the fact that modules do not share type-information. This is particularly a problem when working with C++ (as described next).

Why doesn't C++ inheritance work between modules?

Consider for a moment the following two interface files :

```
// File : a.i
%module a

// Here is a base class
class a {
public:
    a();
    ~a();
    void foo(double);
};

// File : b.i
%module b
```

11 Advanced Topics

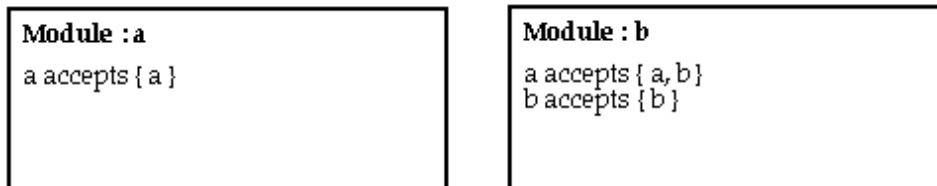
```
// Here is a derived class
%extern a.i // Gets definition of base class

class b : public a {
public:
    bar();
};
```

When compiled into two separate modules, the code does not work properly. In fact, you get a type error such as the following :

```
[beazley@guinness shadow]$ python
Python 1.4 (Jan 16 1997) [GCC 2.7.2]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from a import *
>>> from b import *
>>> # Create a new "b"
>>> b = new_b()
>>> # Call a function in the base class
...
>>> a_foo(b,3)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: Type error in argument 1 of a_foo. Expected _a_p.
>>>
```

However, from our class definitions we know that "b" is an "a" by inheritance and there should be no type-error. This problem is directly due to the lack of type-sharing between modules. If we look closely at the module modules created here, they look like this :



The type information listed shows the acceptable values for various C datatypes. In the "a" module, we see that "a" can only accept instances of itself. In the "b" module, we see that "a" can accept both "a" and "b" instances—which is correct given that a "b" is an "a" by inheritance.

Unfortunately, this problem is inherent in the method by which SWIG makes modules. When we made the "a" module, we had no idea what derived classes might be used at a later time. However, it's impossible to produce the proper type information until after we know all of the derived classes. A nice problem to be sure, but one that can be fixed by making all modules share a single copy of the SWIG run-time library.

The SWIG runtime library

To reduce overhead and to fix type-handling problems, it is possible to share the SWIG run-time functions between multiple modules. This requires the use of the SWIG runtime library which is optionally built during SWIG installation. To use the runtime libraries, follow these steps :

1. Build the SWIG run-time libraries. The `SWIG1.1/Runtime` directory contains a makefile for doing this. If successfully built, you will end up with 6 files that are usually installed in `/usr/local/lib`.

```
libswigtcl.a          # Tcl library (static)
libswigtcl.so         # Tcl library (shared)
libswigpl.a          # Perl library (static)
libswigpl.so         # Perl library (shared)
libswigpy.a          # Python library (static)
libswigpy.so         # Python library (shared)
```

Note that certain libraries may be missing due to missing packages or unsupported features (like dynamic loading) on your machine.

2. Compile all SWIG modules using the `-c` option. For example :

```
% swig -c -python a.i
% swig -c -python b.i
```

The `-c` option tells SWIG to omit runtime support. It's now up to you to provide it separately—which we will do using our libraries.

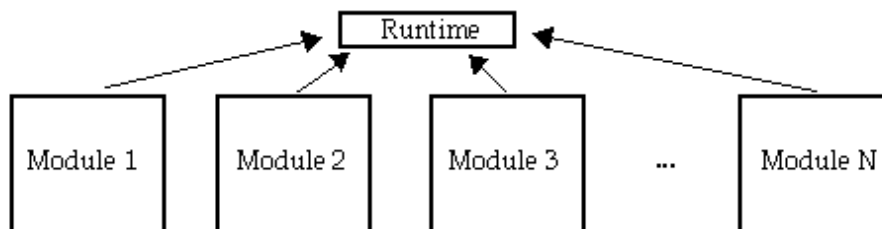
3. Build SWIG modules by linking against the appropriate runtime libraries.

```
% swig -c -python a.i
% swig -c -python b.i
% gcc -c a_wrap.c b_wrap.c -I/usr/local/include
% ld -shared a_wrap.o b_wrap.o -lswigpy -o a.so
```

or if building a new executable (static linking)

```
% swig -c -tcl -ltclsh.i a.i
% gcc a_wrap.c -I/usr/local/include -L/usr/local/lib -ltcl -lswigtcl -lm -o mytclsh
```

When completed you should now end up with a collection of modules like this :



In this configuration, the runtime library manages all datatypes and other information between modules. This management process is dynamic in nature—when new modules are loaded, they contribute information to the run-time system. In the C++ world, one could incrementally load classes as needed. As this process occurs, type information is updated and base-classes learn about derived classes as needed.

A few dynamic loading gotchas

When working with dynamic loading, it is critical to check that only one copy of the run-time library is being loaded into the system. When working with .a library files, problems can sometimes occur so there are a few approaches to the problem.

1. Rebuild the scripting language executable with the SWIG runtime library attached to it. This is actually, fairly easy to do using SWIG. For example :

```
%module mytclsh
%{

static void *__embedfunc(void *a) { return a};
%}

void *__embedfunc(void *);
#include tclsh.i
```

Now, run SWIG and compile as follows :

```
% swig -c -tcl mytclsh.i
% gcc mytclsh_wrap.c -I/usr/local/include -L/usr/local/lib -ltcl -lswigtcl -ldl -lm \
    -o tclsh
```

This produces a new executable "tclsh" that contains a copy of the SWIG runtime library. The weird `__embedfunc()` function is needed to force the functions in the runtime library to be included in the final executable.

To make new dynamically loadable SWIG modules, simply compile as follows :

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

Linking against the `swigtcl` library is no longer necessary as all of the functions are now included in the `tclsh` executable and will be resolved when your module is loaded.

2. Using shared library versions of the runtime library

If supported on your machine, the runtime libraries will be built as shared libraries (indicated by a `.so`, `.sl`, or `.dll` suffix). To compile using the runtime libraries, your link process should look something like this :

```
% ld -shared swigtcl_wrap.o -o libswigtcl.so           # Irix
% gcc -shared swigtcl_wrap.o -o libswigtcl.so         # Linux
% ld -G swigtcl_wrap.o -o libswigtcl.so               # Solaris
```

In order for the `libswigtcl.so` library to work, it needs to be placed in a location where the dynamic loader can find it. Typically this is a system library directory (ie. `/usr/local/lib` or `/usr/lib`).

When running with the shared library version, you may get error messages such as the following


```
Unable to locate libswigtcl.so
```

This indicates that the loader was unable to find the shared library at run-time. To find shared libraries, the loader looks through a collection of predetermined paths. If the `libswigtcl.so` file is not in any of these directories, it results in an error. On most machines, you can change the loader search path by changing the Unix environment variable `LD_LIBRARY_PATH`. For example :

```
% setenv LD_LIBRARY_PATH ./home/beazley/packages/lib
```

A somewhat better approach is to link your module with the proper path encoded. This is typically done using the `-rpath` or `-R` option to your linker (see the man page). For example :

```
% ld -shared example_wrap.o example.o -rpath /home/beazley/packages/lib \
    -L/home/beazley/packages/lib -lswigtcl.so -o example.so
```

The `-rpath` option encodes the location of shared libraries into your modules and gets around having to set the `LD_LIBRARY_PATH` variable.

If all else fails, pull up the man pages for your linker and start playing around.

Dynamic Loading of C++ modules

Dynamic loading of C++ modules presents a special problem for many systems. This is because C++ modules often need additional supporting code for proper initialization and operation. Static constructors are also a bit of a problem.

While the process of building C++ modules is, by no means, an exact science, here are a few rules of thumb to follow :

- Don't use static constructors if at all possible (not always avoidable).
- Try linking your module with the C++ compiler using a command like ``c++ -shared'`. This often solves a lot of problems.
- Sometimes it is necessary to link against special libraries. For example, modules compiled with g++ often need to be linked against the `libgcc.a`, `libg++.a`, and `libstdc++.a` libraries.
- Read the compiler and linker man pages over and over until you have them memorized (this may not help in some cases however).
- Search articles on Usenet, particularly in `comp.lang.tcl`, `comp.lang.perl`, and `comp.lang.python`. Building C++ modules is a common problem.

The SWIG distribution contains some additional documentation about C++ modules in the Doc directory as well.

Inside the SWIG type-checker

The SWIG runtime type-checker plays a critical role in the correct operation of SWIG modules. It not only checks the validity of pointer types, but also manages C++ inheritance, and performs proper type-casting of pointers when necessary. This section provides some insight into what it does, how it works, and why it is the way

it is.

Type equivalence

SWIG uses a name-based approach to managing pointer datatypes. For example, if you are using a pointer like `"double *"`, the type-checker will look for a particular string representation of that datatype such as `"_double_p"`. If no match is found, a type-error is reported.

However, the matching process is complicated by the fact that datatypes may use a variety of different names. For example, the following declarations

```
typedef double   Real;
typedef Real *   RealPtr;
typedef double   Float;
```

define two sets of equivalent types :

```
{double, Real, Float}
{RealPtr, Real *}
```

All of the types in each set are freely interchangeable and the type-checker knows about the relationships by managing a table of equivalences such as the following :

```
double    => { Real, Float }
Real      => { double, Float }
Float     => { double, Real }
RealPtr   => { Real * }
Real *    => { RealPtr }
```

When you declare a function such as the following :

```
void foo(Real *a);
```

SWIG first checks to see if the argument passed is a `"Real *"`. If not, it checks to see if it is any of the other equivalent types (`double *`, `RealPtr`, `Float *`). If so, the value is accepted and no error occurs.

Derived versions of the various datatypes are also legal. For example, if you had a function like this,

```
void bar(Float ***a);
```

The type-checker will accept pointers of type `double ***` and `Real ***`. However, the type-checker does not always capture the full-range of possibilities. For example, a datatype of `'RealPtr **'` is equivalent to a `'Float ***'` but would be flagged as a type error. If you encounter this kind of problem, you can manually force SWIG to make an equivalence as follows:

```
// Tell the type checker that 'Float_ppp' and 'RealPtr_pp' are equivalent.
%init %{
    SWIG_RegisterMapping("Float_ppp", "RealPtr_pp", 0);
}
```

```
%}
```

Doing this should hardly ever be necessary (I have never encountered a case where this was necessary), but if all else fails, you can force the run-time type checker into doing what you want.

Type-equivalence of C++ classes is handled in a similar manner, but is encoded in a manner to support inheritance. For example, consider the following classes hierarchy :

```
class A { };
class B : public A { };
class C : public B { };
class D {};
class E : public C, public D {};
```

The type-checker encodes this into the following sets :

```
A => { B, C, E }           "B isa A, C isa A, E isa A"
B => { C, E }               "C isa B, E isa B"
C => { E }                  "E isa C"
D => { E }                  "E isa D"
E => { }
```

The encoding reflects the class hierarchy. For example, any object of type "A" will also accept objects of type B,C, and E because these are all derived from A. However, it is not legal to go the other way. For example, a function operating on a object from class E will not accept an object from class A.

Type casting

When working with C++ classes, SWIG needs to perform proper typecasting between derived and base classes. This is particularly important when working with multiple inheritance. To do this, conversion functions are created such as the following :

```
void *EtoA(void *ptr) {
    E *in = (E *) ptr;
    A *out = (A *) in;      // Cast using C++
    return (void *) out;
}
```

All pointers are internally represented as void *, but conversion functions are always invoked when pointer values are converted between base and derived classes in a C++ class hierarchy.

Why a name based approach?

SWIG uses a name-based approach to type-checking for a number of reasons :

- One of SWIG's main uses is code development and debugging. In this environment, the type name of an object turns out to be a useful piece of information in tracking down problems.

- In languages like Perl, the name of a datatype is used to determine things like packages and classes. By using datatype names we get a natural mapping between C and Perl.
- I believe using the original names of datatypes is more intuitive than munging them into something completely different.

An alternative to a name based scheme would be to generate type–signatures based on the structure of a datatype. Such a scheme would result in perfect type–checking, but I think it would also result in a very confusing scripting language module. For this reason, I see SWIG sticking with the name–based approach—at least for the foreseeable future.

Performance of the type–checker

The type–checker performs the following steps when matching a datatype :

1. *Check a pointer against the type supplied in the original C declaration. If there is a perfect match, we're done.*
2. *Check the supplied pointer against a cache of recently used datatypes.*
3. *Search for a match against the full list of equivalent datatypes.*
4. *If not found, report an error.*

Most well–structured C codes will find an exact match on the first attempt, providing the best possible performance. For C++ codes, it is quite common to be passing various objects of a common base–class around between functions. When base–class functions are invoked, it almost always results in a miscompare (because the type–checker is looking for the base–type). In this case, we drop down to a small cache of recently used datatypes. If we've used a pointer of the same type recently, it will be in the cache and we can match against it. For tight loops, this results in about 10–15% overhead over finding a match on the first try. Finally, as a last resort, we need to search the internal pointer tables for a match. This involves a combination of hash table lookup and linear search. If a match is found, it is placed into the cache and the result returned. If not, we finally report a type–mismatch.

As a rule of thumb, C++ programs require somewhat more processing than C programs, but this seems to be avoidable. Also, keep in mind that performance penalties in the type–checker don't necessarily translate into big penalties in the overall application. Performance is most greatly affected by the efficiency of the target scripting language and the types of operations your C code is performing.

SWIG 1.1 – Last Modified : Mon Aug 4 10:47:13 1997

SWIG Users Manual

Version 1.1

June, 1997

Copyright(C) 1996, 1997

All Rights Reserved

David M. Beazley

Department of Computer Science

University of Utah

Salt Lake City, Utah 84112 beazley@cs.utah.edu

This document may be freely distributed in whole or part provided this copyright notice is retained. Commercial distribution of this document is prohibited without the express written consent of the author.

SWIG 1.1 is Copyright (C) 1995–1997 by the University of Utah and the University of California and distributed under the following license.

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

Permission is hereby granted, without written agreement and without license or royalty fees, to use, copy, modify, and distribute this software and its documentation for any purpose, provided that (1) The above copyright notice and the following two paragraphs appear in all copies of the source code and (2) redistributions including binaries reproduces these notices in the supporting documentation. Substantial modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated in all files where they apply.

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

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

beazley@cs.utah.edu

Last Modified, August 3, 1997

