SWIG Users Manual

Version 1.1
June 1997

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

Table of Contents

Preface

Introduction

SWIG is a tool for solving problems.

More specifically, SWIG is a simple tool for building interactive C, C++, or Objective-C programs with common scripting languages such as Tcl, Perl, and Python. Of course, more importantly, SWIG is a tool for making C programming more enjoyable and promoting laziness (an essential feature). SWIG is not part of an overgrown software engineering project, an attempt to build some sort of monolithic programming environment, or an attempt to force everyone to rewrite all of their code (ie. code reuse). In fact, none of these things have ever been a priority.

SWIG was originally developed in the Theoretical Physics Division at Los Alamos National Laboratory for building interfaces to large materials science research simulations being run on the Connection Machine 5 supercomputer. In this environment, we were faced with the problems of working with huge amounts of data, complicated machines, and constantly changing code. As scientists, we needed a mechanism for building interactive programs that was extremely easy to use, could keep pace with code that was constantly changing, and didn't get in the way of the real problems that were being solved. Mainly, we just wanted to "cut the crap" and work on the real problems at hand.

While SWIG was originally developed for scientific applications, it has become a general purpose tool that is being used in an increasing variety of other computing applications--in fact almost anything where C programming is involved. Some of the application areas that I am aware of include scientific applications, visualization, databases, semiconductor CAD, remote sensing and distributed objects. Development has been pragmatic in nature--features have been added to address interesting problems as they arise. Most of the really cool stuff has been contributed or suggested by SWIG's users. There has never really been a "grand" design to SWIG other than the goal of creating a practical programming tool that could be used in other applications.

SWIG resources

The official location of SWIG related material is

```
http://www.cs.utah.edu/~beazley/SWIG
```

This site contains the latest version of the software, users guide, and information regarding bugs, installation problems, and implementation tricks. The latest version of the software and related files are also available via anonymous ftp at

```
ftp://ftp.cs.utah.edu/pub/beazley/SWIG
```

You can also subscribe to the SWIG mailing list by sending a message with the text "subscribe swig" to

```
majordomo@cs.utah.edu
```

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

About this manual

This manual has been written in parallel with the development of SWIG because I hate black boxes and I don't like using software that is poorly documented. This manual attempts to describe all aspects of SWIG and how it can be used to solve interesting problems. Don't let the size scare you, SWIG can be quite easy to use. However, covering automatic code generation for four different scripting languages takes abit of explanation. SWIG can do quite a few interesting things that might not be so obvious so I hope that the manual can shed some light on many of these issues. The manual also serves as a general reference describing many of SWIG's implementation issues (I use the manual quite regularly myself).

Prerequisites

This manual assumes that you are interested in writing interactive 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 certainly won't hurt. No prior experience with building C extensions to these languages is required---after all, this is what SWIG allows you to do automatically. However, I do assume that you are 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 (although not significantly more complex).

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 skip right to that chapter and find just about everything you need to know. So, in a sense, you are really getting 3 or 4 manuals in one.

How to avoid reading the manual

If you hate reading manuals, glance at the "Introduction" which contains a few simple examples and the overall philosophy. These examples contain about 95% of everything you need to know to use SWIG. After that, simply use the language-specific chapters for reference. The SWIG distribution also comes with a large directory of examples that illustrate how to do most kinds of things.

Credits

This work would certainly not be possible without the support of many people. I would like to acknowledge Peter Lomdahl, Brad Holian, Shujia Zhou, Niels Jensen, and Tim Germann at Los Alamos National Laboratory for allowing me to pursue this project and for being 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.

I'd also like to acknowledge Chris Johnson and the Scientific Computing and Imaging Group at the University of Utah for their continued support. 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. I'd also like to thank Kevin Butler for his valuable input and contribution of a Windows NT port. Finally, I'd like to acknowledge all of the users who have braved the first few releases and have been instrumental in suggesting way to make SWIG more fun to use than I ever imagined possible.

What's new?

The following significant features are new in version 1.1

- Support for typemaps (a mechanism for customizing SWIG).
- Multiple inheritance.
- Default/optional arguments.
- Perl5 shadow classes.
- Tcl8.0 module (uses the native Tcl8 object interface).
- An entirely new documentation system.
- Limited support for nested structures.
- New object oriented Tcl interface.
- User defined exception handling.
- New directives for better library support (%inline, %extern %import)
- Objective-C support.
- Support for Windows-NT and Macintosh.
- Lots of minor bug fixes to almost everything.

This release should be backwards compatible with interface files generated for SWIG 1.0. However, many things have changed in the SWIG C++ API so special purpose SWIG C++ extensions written for 1.0 will need to be modified.

Bug reports

While every attempt has been made to make SWIG bug-free, occasionally bugs will arrise. To report a bug, send mail to the SWIG mailing list at `swig@cs.utah.edu`. In your message, 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. I attempt to respond to all bug reports, but I can only fix bugs if I know about them.

SWIG is free

SWIG is a completely free package that you can use in any manner that you wish, including modification, redistribution, and use in commercial products. The only restriction on its use is that redistributions of the SWIG compiler should reproduce the SWIG copyright notice in the supporting documentation. The code generated by SWIG can be redistributed in any manner. On a more personal note, if you've used SWIG to make something cool, I'd like to find out more about it so that I can make SWIG better (and to satisfy my curiosity).

Introduction 1

What is SWIG?

SWIG is a code development tool that makes it possible to quickly build powerful scripting language interfaces to C, C++, or Objective-C programs[1]. In a nutshell, SWIG is a compiler that takes C declarations and turns them into the "glue" needed to access them from common scripting languages including Perl, Python, and Tcl. SWIG usually requires no modifications to existing C code and can often be used to build a working interface in a matter of minutes. This makes it possible to do a number of interesting things including :

- Building powerful interfaces to existing C programs.
- Rapid prototyping and application development.
- Interactive debugging.
- Making a graphical user interface (using Tk for example).
- Powerful 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.

There are some computer scientists who seem to believe that the only way to solve complicated problems is to create software of epic proportions and to have some sort of "grand" software design. Unfortunately this seems to lead to solutions that are even more complicated than the original problem. This, in turn enables the user to forget about the original problem and spend their time cursing at their machine (hence, the term "enabling technology"). SWIG, on the other hand, was developed because I was fed up with how much time I was wasting trying to develop flexible scientific applications. I wanted a tool that would let me use scripting languages to glue different things together, but didn't get in the way of the real problems I was working on. I wanted a simple tool that scientists and engineers could use to put together applications involving number crunching, data analysis, and visualization without having to worry about tedious systems programming, making substantial modifications to existing code, trying to figure out a big monolithic computing "environment," or having to get a second degree in computer science. In short, I wanted to have a tool that would improve application development, but stay out of the way as much as possible.

Life before SWIG

SWIG was developed to make my life easier as a C programmer. While C is great for high perfor-

1. For the purposes of documentation, a "C-program" means either ANSI C, C++, or Objective-C.

mance and systems programming, trying to make an interactive and highly flexible C program is a nightmare (in fact, it's much worse than that). The real problem is that for every C program I wrote, I needed to have some sort of interface, but being more interested in other problems, I would always end up writing a really bad interface that was hard to extend, hard to modify, and hard to use. I suppose I could have tried to do something fancy using X11, but who has time to waste weeks or months trying to come up with an interface that is probably going to end up being larger than the original C code? There are more interesting problems to work on.

The real problem, perhaps, is that most C programs end up being structured as follows :

- A collection of functions and variables.
- A `main()` program that starts everything up.
- A bunch of hacks added to make it usable.

The `main()` program may be written to handle command line arguments or to read data from `stdin`, but either way, 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, your C program turns into a hideous unintelligible mess that is even more difficult to modify than it was before (Of course, if you're lucky, your project starts out as an unintelligible mess).

Life after SWIG

With SWIG, I was hoping to avoid many of the headaches of working with C programs, by structuring things as follows :

- A collection of functions and variables.
- A nice interpreted interface language that can be used to access everything.

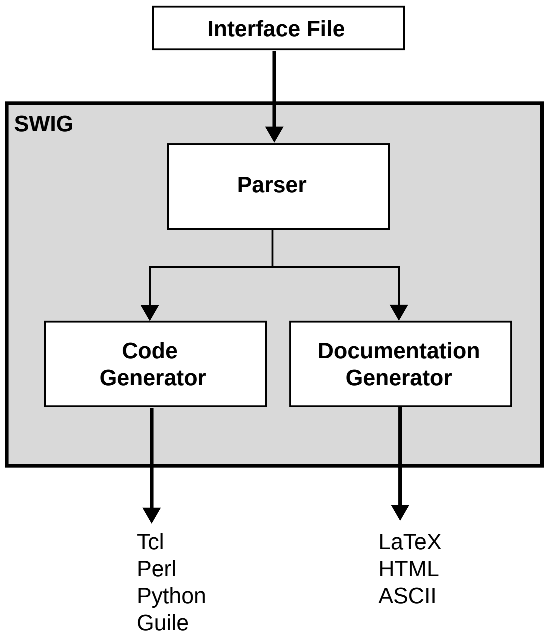
With this model, you keep all of the functionality of your C program, but access it through a scripting language interface instead of writing more C code. This is nice because you are given full freedom to call functions in any order, access variables, and write scripts to do all sorts of interesting things. If you want to change something, just modify a script, and rerun it. If you're trying to debug a collection of functions, you can call them individually and see what they do. If you're trying to build a package out of various components, you can just glue everything together with a scripting language and have a common interface to all of the various pieces.

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.

I like this model of programming. While it's certainly no "silver-bullet", it has made programming alot more enjoyable and has enabled us to solve complicated problems that would have been unneccessarily difficult using only C.

The SWIG package

SWIG is a compiler that takes ANSI C declarations and turns them into a file containing the C code for binding C functions, variables, and constants to a scripting language (SWIG also supports C++ class and Objective-C interface definitions). Input is specified in the form of an “interface file” containing declarations (input can also be given from C source files provided they are sufficiently clean). The SWIG parser takes this input file and passes it on to a code generation and documentation module. These modules produce an interface for a particular scripting language along with a document describing the interface that was created. Different scripting languages are supported by writing new back-end modules to the system.



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. We 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 functions.

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 and used. When loaded, Tcl will now have our new functions and variables added to it. Taking a careful look at the file `example_wrap.c` reveals a hideous mess, but fortunately 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       # This is for Solaris
unix > perl5.003
use example;
```

```
print example::fact(4), "\n";
print example::my_mod(23,7), "\n";
print $example::My_variable + 4.5, "\n";
<ctrl-d>
24
2
7.5
unix >
```

Building a Python module

Finally, let's build a module for Python1.4 (shown for Irix).

```
unix > swig -python example.i
Generating wrappers for Python
unix > gcc -c example.c example_wrap.c -I/usr/local/include/python1.4
unix > ld -shared example.o example_wrap.o -o examplemodule.so     # Irix
unix > Python
Python 1.4 (Sep 10 1996)  [GCC 2.7.0]
Copyright 1991-1996 Stichting Mathematisch Centrum,
Amsterdam
>>> 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, we can often do without it. For example, we 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 also supports a limited form of conditional compilation. If we wanted to make a combination SWIG/C header file, we might do 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);
```

Documentation generation

In addition to producing an interface, SWIG also produces documentation. For our simple example, the documentation file may look like this :

```
example_wrap.c

[ Module : example, Package : example ]

$My_variable
        [ Global : double My_variable ]

fact(n);
        [ returns int  ]

my_mod(n,m);
        [ returns int  ]

get_time();
        [ returns char * ]
```

C comments can be used to provide additional descriptions. SWIG can even grab these out of C source files in a variety of ways. For example, if we process `example.c` as follows :

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

We will get a documentation file that looks like this (with our C comments added) :

```
example_wrap.c

[ Module : example, Package : example ]


$My_variable
        [ Global : double My_variable ]

fact(n);
        [ returns int  ]
        Compute factorial of n

my_mod(n,m);
        [ returns int  ]
        Compute n mod 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, I hope to make SWIG easy to use with most C programs, easy to learn, and easy to remember. Other tools tend to use a precise interface definition language, but I personally find this approach to be painful. When I want to build an interface to a collection of several hundred C functions, I don't necessarily want to write a special interface definition for each one. Nor do I want to have to go dig up the manual because I can't remember the syntax.

On the other hand, using C syntax can be ambiguous. For example, if you have the following declaration

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

We don't really 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 programming, 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

(C++ templates for example). Thus, SWIG may not recognize advanced C/C++ constructs. I make no apologies for this--SWIG was designed to access C, but was never intended to magically turn scripting languages into some sort of bizarre C++ interpreter. 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 I intended, SWIG requires minimal modification to existing C code. This makes SWIG extremely easy to use with existing packages, but also 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. In a similar spirit, I don't believe that there is any one "best" scripting language--use whichever one you like (they're all pretty good). There's no real reason why a particular application couldn't support multiple scripting language interfaces to serve different needs (or to have no scripting language interface at all).

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. This is my goal.

Event driven C programming

By adding a scripting language interface to a program, SWIG encourages an event-driven style of programming (although it may not be immediately obvious). An event driven program basically consists of a large-collection of functions (called callback functions), and an infinite loop that just sits and waits for the user to do something. When the user does something like type a command, hit a key, or move the mouse the program will call a function to perform an operation--this is an event. Of course, unless you've been living in cave for the last 20 years, you've used this kind of approach when running any sort of graphical user interface.

While you may not be using SWIG to develop a GUI, the underlying concept is the same. The scripting language acts as an event-loop waiting for you to issue commands. Unlike a traditional C application (that may use command line options), there is no longer a fixed sequence of operations. Functions may be issued in any order and at any time. Of course, this flexibility is exactly what we want!

However, there are a number of things to keep in mind when developing an application with SWIG :

- Functions may be called at any time and in any order. It is always a good idea for functions to check their arguments and internal state to see if it's legal to proceed (Not doing so usually results in mysterious crashes later on).
- Code should be structured as a collection of independent modules, not a huge mess of

interrelated functions and variables (ie. spaghetti code).
- Global variables should be used with care.
- Careful attention to the naming of variables and functions may be required to avoid namespace conflicts when combining packages.

While it may be hard (or impossible) to address all these problems in a legacy code, I believe that using SWIG encourages all of the above qualities when developing new applications. This results in code that is more reliable, more modular, and easier to integrate into larger packages. By providing a non-intrusive, easy to use tool, it is possible to develop highly reliable event-driven code from the start--not as a hack to be added later. As a final sales pitch, in the initial application for which SWIG was developed, code reliability and flexibility has increased substantially while code size has decreased by more than 25%. I believe this is a good thing.

Automatic documentation generation

SWIG makes it very easy to build large interactive C programs, but it can sometimes be hard to remember what C functions are available and how they are called in the scripting language interface. To address this problem, SWIG automatically generates a documentation file in a number of different formats. C comments can be used to provide additional descriptions of each function, and documentation can be grouped into a hierarchy of sections and subsections. The documentation file is intended to provide a reasonable description of the scripting language interface. While it's no competition for a full-blown C code documentation generator, the documentation system can do a reasonable job of documenting an interface.

Summary

At this point, you know about 95% of everything you need to know to start using SWIG. First, functions and variables are specified using ANSI C, C++, or Objective-C syntax. These may appear in a separate interface file or you can use a C source file (if it is sufficiently clean). SWIG requires no modifications to existing C code so it's easy to get started. To build a module, use the `swig` command with an appropriate target language option. This generates a C file that you need to compile with the rest of your code and you're ready to go.

I don't consider there to be a "right" or "wrong" way to use SWIG, although I personally use separate interface files for a variety of reasons :

- It helps keep me organized.
- It's usually not necessary to wrap every single C function in a program.
- SWIG provides a number of directives that I tend to use alot.
- Having a special interface file makes it clear where the scripting language interface is defined. If you decide to change something in the interface, it's easy to track down if you have special file.

Of course, your mileage may vary.

SWIG for Windows and Macintosh

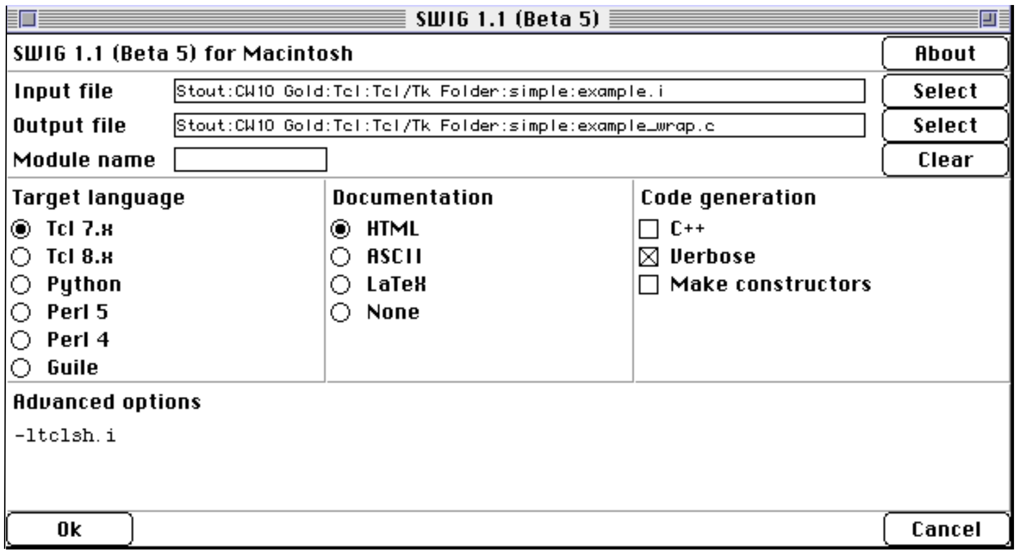
SWIG has been primarily developed and designed to work with Unix-based applications. However, most of the scripting languages supported by SWIG are available on a variety of other platforms including Windows 95/NT and Macintosh. SWIG generated code is mostly compatible with these versions (and SWIG itself can now run on these platforms).

SWIG on Windows 95/NT

The Windows 95/NT port of SWIG (provided by Kevin Butler), is a straightforward translation of the Unix version. At this time it is only known to work with Microsoft Visual C++ 4.x, but there is nothing to prevent its use with other compilers (at least not that I'm aware of). SWIG should be invoked from the MS-DOS prompt in exactly the same manner as in Unix. SWIG can also be used in NMAKE build files and added to Visual C++ projects under the custom build setting. As of this writing, SWIG is known to work with Windows versions of Tcl/Tk, Python, and Perl (including the ActiveWare Perl for Win32 port). SWIG makes extensive use of long filenames, so it is unlikely that the SWIG compiler will operate correctly under Windows 3.1 or DOS (the wrapper code generated by SWIG may compile however).

SWIG on the Power Macintosh

A Macintosh port of SWIG is also available, but is highly experimental at this time. It only works on PowerPC based Macintosh systems running System 7 or later. Modules can be compiled with the Metrowerks Code Warrior compiler, but support for other compilers is unknown. Due to the lack of command line options on the Mac, SWIG has been packaged in a Tcl/Tk interface that allows various settings and command line options to be specified with a graphical user interface. Underneath the hood, SWIG is identical to the Unix/Windows version. It recognizes the same set of options and generates identical code. Any SWIG command line option listed in this manual can be given to the Mac version under "Advanced Options" shown in the figure. At this writing, SWIG is only known to support Mac versions of Tcl/Tk. Work on Macintosh versions of Python and Perl is underway.



SWIG 1.1 (Beta 5) for Macintosh

Input file: Stout:CW10 Gold:Tcl:Tcl/Tk Folder:simple:example.i

Output file: Stout:CW10 Gold:Tcl:Tcl/Tk Folder:simple:example_wrap.c

Module name:

Target language	Documentation	Code generation
● Tcl 7.x	● HTML	☐ C++
○ Tcl 8.x	○ ASCII	☒ Verbose
○ Python	○ LaTeX	☐ Make constructors
○ Perl 5	○ None	
○ Perl 4		
○ Guile		

Advanced options

```
-ltclsh.i
```

Cross platform woes

While SWIG and various freely available scripting languages are supported on different platforms, developing cross platform applications with these tools is still immature and filled with pitfalls. In fact, it's probably only recommended for true masochists. Despite this, it's interesting to think about using freely available tools to provide common interfaces to C/C++ code. I believe that SWIG may help, but it is by no means a cross-platform solution by itself.

How to survive this manual

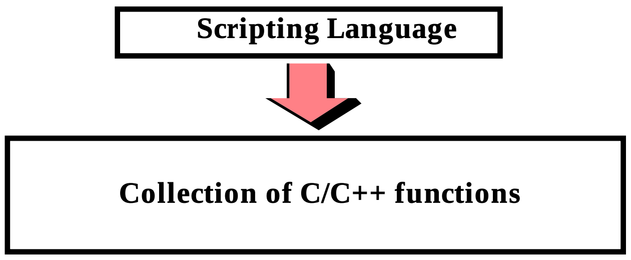
This manual was written to support the Unix version of SWIG. However, all of the main concepts and usage of SWIG also apply to the Windows and Macintosh versions. You should be forewarned that most of the examples are Unix-centric and may not compile correctly on other machines. However, most of the examples provided with the SWIG distribution have now been tested under both Unix and Windows-NT. When applicable, I will try to point out incompatibilities, but make no promises...

Scripting Languages 2

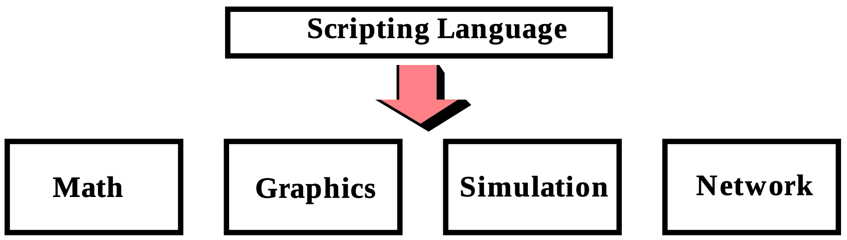
SWIG is all about using scripting languages with C/C++ to make flexible applications. This chapter provides a brief overview of several concepts and important aspects of this interface. Many of SWIG's potential users may not have considered using a scripting language before, so I hope that this chapter can provide a little motivation.

The two language view of the world

By developing SWIG, I am trying to build systems that are loosely structured as follows :



A real application might look more like this :



In either case, we are interested in controlling a C/C++ program with a scripting language interface. Our interface may be for a small group of functions or a large collection of C libraries for performing a variety of tasks. In this model, C functions are turned into commands. To control the program, the user now types these commands or writes scripts to perform a particular operation. If you have used commercial packages such as MATLAB or IDL, it is a very similar model--you execute commands and write scripts, yet most of the underlying functionality is still written in C or Fortran for performance.

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 lists, arrays, and hash tables.

Will scripting languages make my C program inefficient?

One of the criticisms of scripting languages is that they are interpreted and slow. No doubt about it, a scripting language will always run much slower than C. However, if you are using a scripting language to control a big C program, most of your functionality is still written in C and still fast. Thus, there is really no difference between writing the following in C

```
for (i = 0; i < 1000; i++) {
      call a bunch of C functions to do something
}
```

or writing the same thing in Python :

```
for i in range(0,1000):
      call a bunch of C functions to do something
```

Most of the time is still spent in the underlying C functions. Of course, you wouldn't want to write the inner loop of a matrix multiply in a scripting language, but you already knew this. It is also worth noting that reimplementing certain operations in C might not lead to better performance. For example, Perl is highly optimized for text-processing operations. Most of these operations are already implemented in C (underneath the hood) so in certain cases, using a scripting language may actually be faster than an equivalent implementation in C.

Will adding a scripting language to my C program make it unmanagable?

A fear among some users is that by adding a second language, you will end up with a package that is hard to maintain and use. I believe that there are two answers to this question. If you find yourself modifying the C code to fit it into a specific scripting language, then it will be difficult to maintain. By doing this, you will lock yourself into a particular language. If that language changes or disappears off the face of the earth, then you will be left with serious maintenance problems. On the flip side of the coin, a non-invasive tool like SWIG can build interfaces without requiring language-specific modifications to the underlying C code. If the scripting language changes, it is easy to update the resulting interface. If you decide that you want to scrap the whole interface scheme and try something else, you still have a clean set of C libraries

My personal experience has been that adding a scripting language to a C program makes the C program more managable! You are encouraged to think about how your C program is structured and how you want things to work. In every program in which I have added a scripting interface, the C code has actually decreased in size, improved in reliability, become easier to maintain, while becoming more functional and flexible than before.

How does a scripting language talk to C?

Scripting languages are built around a small parser that reads and executes statements on the fly as your program runs. Within the parser, there is a mechanism for executing commands or accessing variables. However, in order to access C functions and variables, it is necessary to tell the parser additional information such as the name of the function, what kind of arguments does it take, and what to do when it is called. Unfortunately, this process can be extremely tedious and technical. SWIG automates the process and allows you to forget about it. In any case, it's probably a good idea to know what's going on under the hood.

Wrapper functions

Suppose you have an ordinary C function like this :

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

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

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

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

```
int wrap_fact(ClientData clientData, Tcl_Interp *interp,
              int argc, char *argv[]) {
       int _result;
       int _arg0;
       if (argc != 2) {
              interp->result = "wrong # args";
              return TCL_ERROR;
       }
       _arg0 = atoi(argv[1]);
       _result = fact(_arg0);
       sprintf(interp->result,"%d", _result);
       return TCL_OK;
}
```

Once we have created a wrapper function, the final step is to tell the scripting language about our new function. This is usually done in an initialization function called by the language when our module is loaded. For example, adding the above function to the Tcl interpreter would require 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.

While 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 is a slightly more difficult problem. The idea here is to map a C/C++ global variable into a variable in the scripting language (we are "linking" a variable in the scripting language to a C variable). For example, if you have the following variable:

```
double My_variable = 3.5;
```

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

```
$a = $My_variable * 2.3;
```

Unfortunately, the process of linking variables is somewhat problematic and not supported equally in all scripting languages. There are two primary methods for approaching this problem:

- **Direct access**. Tcl provides a mechanism for directly accessing C `int`, `double`, and `char` * datatypes as Tcl variables. Whenever these variables are used in a Tcl script, the interpreter will directly access the corresponding C variable. In order for this to work, one must first "register" the C variables with the Tcl interpreter. While this approach is easy to support it is also somewhat problematic. Not all C datatypes are supported, and having Tcl directly manipulate your variables in its native representation could be potentially dangerous.
- **Access through function calls**. Languages such as Perl and Python can access global variables using a function call mechanism. Rather than allowing direct access, the idea is to provide a pair of set/get functions that set or get the value of a particular variable. In many cases, this mechanism may be completely hidden. For example, it is possible to create a magical Perl variable that looks and feels just like a normal Perl variable, but is really mapped into a C variable via a pair of set/get functions. The advantage of this approach is that it is possible to support almost all C datatypes. The disadvantage is that it introduces alot of complexity to the wrapper code as it is now necessary to write a pair of C functions for every single global variable.

SWIG supports both styles of variable linking although the latter is more common. In some cases, a hybrid approach is taken (for example, the Tcl module will create a pair of set/get functions if it encounters a datatype that Tcl can't support). Fortunately, global variables are relatively rare when working with modular code.

Constants

Constants can easily be created by simply creating a new variable in the target language with the appropriate value. Unfortunately, this can have the undesirable side-effect of making the constant non-constant. As a result, a somewhat better (although perhaps inefficient) method of creating constants is to install them as read-only variables. SWIG tends to prefer this approach.

Structures and classes

Most scripting languages have trouble directly dealing with C structures and C++ classes. This is because the use of structures is inherently C dependent and it doesn't always map well into a scripting language environment. Many of these problems are simply due to data representation issues and differences in the way C and a scripting language might represent integers, floats, strings, and so on. Other times, the problem is deeper than that--for example, what does it mean (if anything) to try and use C++ inheritance from Perl?

Dealing with objects is a tough problem that many people are looking at. Packages such as CORBA and ILU are primarily concerned with the representation of objects in a portable manner. This allows objects to be used in distributed systems, used with different languages and so on. SWIG is not concerned with the representation problem, but rather the problem of accessing and using C/C++ objects from a scripting language (in fact SWIG has even been used in conjunction with CORBA-based systems).

To provide access, the simplist approach is to transform a structure into a collection of accessor functions. 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_y_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);
```

When accessed in Tcl, the functions could be used as follows :

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

The accessor functions provide a mechanism for accessing a real C/C++ object. Since all access occurs though these function calls, Tcl does not need to know anything about the actual representation of a `Vector`. This simplifies matters considerably and steps around many of the problems associated with objects--in fact, it lets the C/C++ compiler do most of the work.

Shadow classes

As it turns out, it is possible to use the low-level accessor functions above to create something known as a “shadow” class. In a nutshell, a “shadow class” is a funny 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). Of course, in reality, it’s just a slick way of accessing objects that is more natural to most programmers. 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 natural manner. For example, in Python, you might do the following,

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

while in Perl5, it might look like this :

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

and 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. However, the introduction of additional "objects" can also produce excessive overhead if working with huge numbers of objects in this manner. Despite this, shadow classes turn out to be extremely useful. The actual implementation is covered later.

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. Unfortunately, this almost always seems to be the most difficult part. There are two fundamental approaches for doing this. First, you can build an entirely new version of the scripting language interpreter with your extensions built into it. Alternatively, you can build a shared library and dynamically load it into the scripting language as needed. Both approachs are described below :

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 executable. When using static linking, SWIG will provide a `main()` program for you so you usually just have to compile as follows (shown for Tcl) :

```
unix > swig -tcl -ltclsh.i example.i
Generating wrappers for Tcl.
unix > gcc example.c example_wrap.c -I/usr/local/include \
       -L/usr/local/lib -ltcl -lm -o my_tclsh
```

`my_tclsh` is a new executable containing the Tcl intepreter. `my_tclsh` will be exactly the same as tclsh except with your new commands added to it. When invoking SWIG, the `-ltclsh.i` option includes support code needed to rebuild the `tclsh` application.

Virtually all machines support static linking and in some cases, it may be the only way to build an extension. The downside to static linking is that you can end up with a large executable. In a very large system, the size of the executable may be prohibitively large.

Shared libraries and dynamic loading

An alternative to static linking is to build a shared library. With this approach, you build a shared object file containing only the code related to your module (on Windows, this is the same as building a DLL). Unfortunately the process of building these modules varies on every single machine, but the procedure for a few common machines is show 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 Irix
gcc -c example.c example_wrap.c -I/usr/local/include
ld -shared example.o example_wrap.o -o example.so

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

When working with C++ codes, the process of building shared libraries is more difficult--primarily due to the fact that C++ modules may need additional code in order to operate correctly. On most 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
```

The advantages to dynamic loading is that you can use modules as they are needed and they can be loaded on the fly. The disadvantage is that dynamic loading is not supported on all machines (although support is improving). The compilation process also tends to be more complicated than what might be used for a typical C/C++ program.

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/data1/beazley/graph/graph.py", line 2, in ?
    import graphc
ImportError:  1101:/home/sci/data1/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.

- Set the UNIX environment variable `LD_LIBRARY_PATH` to the directory where shared libraries are located before running Python.
- 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.

With a little patience and after some playing around, you can usually get things to work. Afterwards, building extensions becomes alot easier.

SWIG Basics 3

Running SWIG

SWIG is invoked by the `swig` command. This command has a number of options including:

```
swig <options> filename

-tcl                  Generate Tcl wrappers
-tcl8                 Generate Tcl 8.0 wrappers
-perl5                Generate Perl5 wrappers
-python               Generate Python wrappers
-perl4                Generate Perl4 wrappers
-guile                Generate Guile wrappers
-dascii               ASCII documentation
-dlatex               LaTeX documentation
-dhtml                HTML documentation
-dnone                No documentation
-c++                  Enable C++ handling
-objc                 Enable Objective-C handling.
-Idir                 Set SWIG include directory
-lfile                Include a SWIG library file.
-c                    Generate raw wrapper code (omit supporting code)
-v                    Verbose mode (perhaps overly verbose)
-o outfile            Name of output file
-d docfile            Set name of documentation file (without suffix)
-module name          Set name of SWIG module
-Dsymbol              Define a symbol
-version              Show SWIG's version number
-help                 Display all options
```

This is only a partial list of options. A full listing of options can be obtained by invoking "`swig -help`". Each target language may have additional options which can be displayed using "`swig -lang -help`" where `-lang` is one of the target languages above.

Input format

As input, SWIG takes a file containing ANSI C/C++ declarations[1]. This file may be a special "interface file" (usually given a .i suffix), a C header file or a C source file. The most common method of using SWIG is with a special interface file. These files contain ANSI C declarations like a header file, but also contain SWIG directives and documentation. Interface files usually have the following format :

1. Older style C declarations are not supported

```
%module mymodule
%{
#include "myheader.h"
%}
// Now list ANSI C variable and function declarations
```

The name of the module (if supplied) must always appear before the first C declaration or be supplied on the SWIG command line using the `-module` option (When the module name is specified on the command line, it will override any module name present in a file). Everything inside the `%{,%}` block is copied verbatim into the resulting output file. The `%{,%}` block is optional, but most interface files use one to include the proper header files.[1]

SWIG Output

By default an interface file with the name `myfile.i` will be transformed into a file called `myfile_wrap.c`. The name of the output file can be changed using the `-o` option. When working with some C++ compilers, the `-o` option can be used to give the output file a proper C++ suffix. The output file usually contains everything that is needed to build a working module for the target scripting language. Compile it along with your C program, link it, and you should be ready to run.

Comments

C and C++ style comments may be placed in interface files, but these are used to support the automatic documentation system. Please see the documentation section for more details on this. Otherwise, SWIG throws out all comments so you can use a C++ style comment even if the resulting wrapper file is compiled with the C compiler.

C Preprocessor directives

SWIG does not run the C preprocessor. If your input file makes extensive use of the C preprocessor, SWIG will probably hate it. However, SWIG does recognize a few C preprocessor constructs that are quite common in C code :

- `#define`. Used to create constants
- `#ifdef,#ifndef,#else,#endif`, `#if, #elif`. Used for conditional compilation

All other C preprocessor directives are ignored by SWIG (including macros created using `#define`).

SWIG Directives

SWIG directives are always preceded by a "%" to distinguish them from normal C directives and declarations. There are about two dozen different directives that can be used to give SWIG hints, provide annotation, and change SWIG's behavior in some way or another.

Limitations in the Parser (and various things to keep in mind)

It was never my intent to write a full C/C++ parser. Therefore SWIG has a number of limitations to keep in mind.

1. Previous versions of SWIG required a `%{,%}` block. This restriction has been lifted in SWIG 1.1.

- Functions with variable length arguments (ie. "...") are not supported.
- Complex declarations such as function pointers and arrays are problematic. You may need to remove these from the SWIG interface file or hide them with a typedef.
- C++ source code (what would appear in a .C file) is especially problematic. Running SWIG on C++ source code is highly discouraged.
- More sophisticated features of C++ such as templates and operator overloading are not supported. Please see the section on using SWIG with C++ for more details. When encountered, SWIG may issue a warning message or a syntax error if it can't figure out you are trying to do.

Many of these limitations may be eliminated in future releases. It is worth noting that many of the problems associated with complex declarations can sometimes be fixed by clever use of `typedef`.

If you are not sure whether SWIG can handle a particular declaration, the best thing to do is try it and see. SWIG will complain loudly if it can't figure out what's going on. When errors occur, you can either remove the offending declaration, conditionally compile it out (SWIG defines a symbol `SWIG` that can be used for this), or write a helper function to work around the problem.

Simple C functions, variables, and constants

SWIG supports just about any C function, variable, or constant involving built-in C datatypes. For example :

```
%module example

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

will create two commands called "`sin`" and "`strcmp`", a global variable "`My_variable`", and two constants "`STATUS`" and "`VERSION`". Things work about like you would expect. For example, in Tcl :

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

The main concern when working with simple functions is SWIG's treatment of basic datatypes. This is described next.

Integers

SWIG maps the following C datatypes into integers in the target scripting language.

```
int
short
long
unsigned
signed
unsigned short
unsigned long
unsigned char
signed char
bool
```

Scripting languages usually only support a single integer type that corresponds to either the `int` or `long` datatype in C. When converting from C, all of the above datatypes are cast into the representation used by the target scripting language. Thus, a 16 bit short in C may be converted to a 32 bit integer. When integers are converted from the scripting language back into C, the value will be cast into the appropriate type. The value will be truncated if it is too large to fit into the corresponding C datatype. This truncation is not currently checked.

The `unsigned char` and `signed char` datatypes are special cases that are treated as integers by SWIG. Normally, the `char` datatype is mapped as an ASCII string.

The `bool` datatype is cast to and from an integer value of 0 and 1.

Some care is in order for large integer values. Most scripting language 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 that may show up as 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 should verify the correct operation of your program after wrapping it with SWIG.

Floating Point

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 `double` except in Tcl 7.x which uses character strings. The rarely used datatype of “long double” is not supported by SWIG.

Character Strings

The `char` datatype is mapped into a NULL terminated ASCII string with a single character. When used in a scripting language it will show up as a tiny string containing the character value. When converting the value back into C, SWIG will take a character string from the scripting language and strip off the first character as the char value. Thus if you try to assigned the value “foo” to a `char` datatype, it will get the value ‘f’.

The `char *` datatype is assumed to be a NULL-terminated ASCII string. SWIG maps this into a character string in the target language. SWIG converts character strings in the target language to NULL terminated strings before passing them into C/C++. It is illegal for these strings to have embedded NULL bytes although there are ways to work around this problem.

The `signed char` and `unsigned char` datatypes are mapped into integer values. The following example illustrates the mapping of `char` datatypes.

```
%{
#include <stdio.h>
#include <ctype.h>
#include <string.h>
signed char sum(signed char a, signed char b) { return a+b;}
%}

int  strcmp(char *, char *);
char toupper(char);
signed char sum(signed char a, signed char b);
```

A Tcl script using these functions (and the resulting output) might be as follows.

```
tclsh > strcmp Mike John
1
tclsh > toupper g
G
tclsh > sum 17 -8
9
```

Variables

SWIG attempts to map C/C++ global variables into scripting language variables. For example:

```
%module example

double foo;
```

will result in a scripting language variable that can be used as follows :

```
# 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
```

Whenever this “special” variable is used, the underlying C global variable will be accessed. As it turns out, working with global variables is one of the most tricky aspects of SWIG. Whenever possible, you should try to avoid the use of globals. Fortunately, most modular programs make limited (or no) use of globals.

Constants

Constants can be created using `#define`, `const`, or enumerations. Constant expressions are also allowed. The following interface file shows a few valid constant declarations :

```
#define I_CONST       5                    // An integer constant
#define F_CONST       3.14159              // A Floating point constant
#define S_CONST       "hello world"        // A string constant
#define NEWLINE       '\n'                 // Character constant
#define MODE          DEBUG                // Sets MODE to DEBUG.
                                           // DEBUG is assumed to be an
                                           // int unless declared earlier
enum boolean {NO=0, YES=1};
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
              SEP, OCT, NOV, DEC};
const double PI 3.141592654;
#define F_CONST2      (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 or can be explicitly set using a cast. For example, a number with a decimal point is assumed to be floating point. When no explicit value is available for a constant, SWIG will use the value assigned by the C compiler. For example, no values are given to the `months` enumeration, but this is no problem---SWIG will use whatever the C compiler picks.

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 to retrieve the correct enum values.

Pointers and complex objects

As we all know, most C programs have much more than just integers, floating point numbers, and character strings. There may be pointers, arrays, structures, and other objects floating around. Fortunately, this is usually not a problem for SWIG.

Simple pointers

Pointers to basic C datatypes such as

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

can be used freely in an interface file. SWIG encodes pointers into a representation containing the actual value of the pointer and a string representing the datatype. Thus, the SWIG representation of the above pointers (in Tcl), might look like the following :

```
_10081012_int_p
_1008e124_double_ppp
_f8ac_char_pp
```

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. 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 write a C program (well, with a few limitations).

The scripting language representation of a pointer should never be manipulated directly (although nothing prevents this). SWIG does not normally map pointers into high-level objects such as associative arrays or lists (for example, it might be desirable to convert an `int *` into an list of integers). There are several reasons for this :

- Adding special cases would make SWIG more complicated and difficult to maintain.
- 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 one million elements, converting it into a Tcl, Perl, or Python list would probably be an extremely bad idea.
- By treating all pointers equally, it is easy to know what you're going to get when you create an interface (pointers are treated in a consistent manner).

As it turns out, you can remap any C datatype to behave in new ways so these rules are not set in stone. Interested readers should look at the chapter on typemaps.

Run time pointer type checking

By allowing pointers to be manipulated interactively in a scripting language, we have effectively bypassed the type-checker in the C/C++ compiler. By encoding pointer values with a datatype, SWIG is able to perform run-time type-checking in order to prevent mysterious system crashes and other anomalies. By default, SWIG uses a strict-type checking model that checks the datatype of all pointers before passing them to C/C++. However, you can change the handling of pointers using the `-strict` option:

```
-strict 0            No type-checking (living on the edge)
-strict 1            Generate warning messages (somewhat annoying)
-strict 2            Strict type checking (the default)
```

Strict type checking is the recommended default since is the most reliable and most closely follows the type checking rules of C. In fact, at this time, the other two modes should be considered to be outdated SWIG features that are supported, but no longer necessary[1].

By default, SWIG allows "NULL" pointers to be passed to C/C++. This has the potential to crash code and cause other problems if you are not careful. Checks can be placed on certain values to

1. In early versions of SWIG, some users would disable the type-checker to work around type-casting problems. This is no longer necessary as most type-related problems can be solved using the pointer.i library file included with SWIG1.1.

prevent this but this requires the use of typemaps (described in later chapters).

Like C, it should also be noted that functions involving `void` pointers can accept any kind of pointer object.

Derived types, structs, and classes

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

All complex datatypes are pointers

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

While all of 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 it's used as a pointer, so it doesn't really matter what it is. If you wrapped this module into Python, you could use the functions just like you would 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.

What happens when SWIG encounters an unknown datatype?

When SWIG encounters an unknown datatype, it automatically assumes that it is some sort of complex datatype. 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 so it will assume that you know what you are doing and map it into a pointer. This makes perfect sense because the underlying C function is using pointers in the first place. Unlike C or C++, SWIG does not actually care whether `Matrix` has been previously defined in the interface file or not. While this may sound strange, it makes it possible for SWIG to generate interfaces from only partial information. In many cases, you may not care what a `Matrix` really is as long as you can pass references to one around in the scripting language interface. The downside to this relaxed approach is that typos may go completely undetected by SWIG[1]. You can also end up shooting yourself in the foot, but presumably you've passed your programming safety course if you've made it this far.

As a debugging tool, SWIG will report a list of used, but undefined datatypes, if you run it with the `-stat` option.

```
[beazley@guinness SWIG1.1b6]$ swig -stat matrix.i
Making wrappers for Tcl
Wrapped 1 functions
Wrapped 0 variables
Wrapped 0 constants
The following datatypes were used, but undefined.
     Matrix
[beazley@guinness SWIG1.1b6]$
```

Typedef

`typedef` can be used to remap datatypes within SWIG. For example :

```
typedef unsigned int size_t;
```

This makes SWIG treat `size_t` like an unsigned int. Use of `typedef` is fairly critical in most applications. Without it, SWIG would consider `size_t` to be a complex object (which would be incorrectly converted into a pointer).

Getting down to business

So far, you know just about everything you need to know to use SWIG to build interfaces. In fact, using nothing but basic datatypes and opaque pointers it is possible to build scripting language interfaces to most kinds of C/C++ packages. However, as the novelty wears off, you will want to do more. This section describes SWIG's treatment of more sophsticated issues.

Passing complex datatypes by value

Unfortunately, not all C programs manipulate complex objects by reference. When encountered, SWIG will transform the corresponding C/C++ declaration to use references instead. For example, suppose you had the following function :

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

SWIG will transform this function call into the equivalent of the following :

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

1. Fortunately, if you make a typo, the C compiler will usually catch it when it tries to compile the SWIG generated wrapper file.

In the scripting language, `dot_product` will now take references to Vectors instead of Vectors, although you may not notice the change.

Return by value

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

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

This function is returning a complex object, yet SWIG only likes to work with references. Clearly, something must be done with the return result, or it will be lost forever. As a result, SWIG transforms this function into the following code :

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

or if using C++ :

```
Vector *wrap_cross(Vector *v1, Vector *v2) {
        Vector *result = new Vector(cross(*v1,*v2)); // Uses default copy constructor
        return result;
}
```

SWIG is forced to create a new object and return a reference to it. It is up to the user to delete the returned object when it is no longer in use. When used improperly, this can lead to memory leaks and other problems. Personally, I'd rather live with a potential memory leak than forbid the use of such a function. Needless to say, some care is probably in order (you need to be aware of this behavior in any case).

Linking to complex variables

When global variables or class members involving complex datatypes are encountered, SWIG converts them into references. For example, a global variable like this :

```
Vector unit_i;
```

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

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

Returning a reference to the variable makes it accessible like any other object of this type. When setting the value, we simply make a copy of some other Vector reference. Again some caution is in order. A global variable created in this manner will show up as a reference in the target scripting language. It would be an extremely bad idea to free or destroy such a reference. Similarly, one can run into problems when copying complex C++ objects in this manner. Fortunately, in well-written modular code, excessive use (or abuse) of global variables is rare.

Arrays

The use of arrays in the current version of SWIG is supported, but with caution. If simple arrays appear, they will be mapped into a pointer representation. Thus, the following declarations :

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

will be processed as if they were declared like this:

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

Multi-dimensional arrays are transformed into a single pointer since `a[][]` and `**a` are not the same thing (even though they can be used in similar ways). Rather, `a[][]` is mapped to *a, where `*a` is the equivalent of `&a[0][0]`. The reader is encouraged to dust off their C book and look at the section on arrays before using them with SWIG.

Be aware that use of arrays may cause compiler warnings or errors when compiling SWIG generated modules. While every attempt has been made to eliminate these problems, handling of arrays can be somewhat problematic due to the subtle differences between an array and a pointer.

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.

Renaming declarations

Normally, the name of a C function is used as the name of the command added to the target scripting language. Unfortunately, this name may conflict with a keyword or already existing function in the scripting language. Naming conflicts can be resolved using the `%name` directive as shown :

```
// interface.i
```

```
%name(my_print) extern void print(char *);
%name(foo) extern int a_really_long_and_annoying_name;
```

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

A more powerful renaming operation can be performed with the `%rename` directive as follows :

```
%rename oldname newname;
```

`%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 print output; // Rename all 'print' functions to 'output'
```

SWIG does not perform any checks to see if the functions it adds 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.

Overriding call by reference

SWIG is quite literal in its interpretation of datatypes. If you give it a pointer, it will use pointers. For example, if you're trying to call a function in a Fortran library (through its C interface) all function parameters will have to be passed by reference. Similarly, some C functions may use pointers in unusual ways. The `%val` directive can be used to change the calling mechanism for a C function. For example :

```
// interface.i
%{
#include <time.h>
%}

typedef long time_t;
time_t time(time_t *t);
struct tm *localtime(%val time_t *t);
char *asctime(struct tm *);
```

The `localtime()` function takes a pointer to a `time_t` value, but we have forced it to take a value instead in order to match up nicely with the return value of the `time()` function. When used in Perl, this allows us to do something like this :

```
$t = time(0);
$tm = localtime($t); # Note passing $t by value here
print $asctime($tm);
```

Internally, the `%val` directive creates a temporary variable. The argument value is stored in this variable and a function call is made using a pointer to the temporary variable. Of course, if the function returns a value in this temporary variable, it will be lost forever.

Default/optional arguments

SWIG allows default arguments to be used in both C/C++ code as follows :

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

To specify a default argument, simply specify it the function prototype as shown. SWIG will generate wrapper code in which the default arguments are optional. 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
```

While the ANSI C standard does not specify default arguments, default arguments used in a SWIG generated interface work with both C and C++.

Pointers to functions

At the moment, the SWIG parser has difficulty handling pointers to functions (a deficiency that is being corrected). However, having function pointers is useful for managing C callback functions and other things. To properly handle function pointers, it is currently necessary to use `typedef.` For example, the function

```
void do_operation(double (*op)(double,double), double a, double b);
```

should be handled as follows :

```
typedef double (*OP_FUNC)(double,double);
void do_operation(OP_FUNC op, double a, double b);
```

SWIG understands both the typedef declaration and the later function call. It will treat `OP_FUNC` just like any other complex datatype. In order for this approach to work, it is necessary that the typedef declaration be present in the original C code--otherwise, the C compiler will complain. If you are building a separate interface file to an existing C program and do not want to make changes to the C source, you can also do the following :

```
// File : interface.i
%typedef double (*OP_FUNC)(double,double);
double do_operation(OP_FUNC op, double a, double b);
```

`%typedef` forces SWIG to generate a `typedef` in the C output code for you. This would allow the interface file shown to work with the original unmodified C function declaration.

Constants containing the addresses of C functions can also be created. For example, suppose you have the following callback functions :

```
extern double op_add(double a, double b);
extern double op_sub(double a, double b);
extern double op_mul(double a, double b);
```

The addresses of these functions could be installed as scripting language constants as follows :

```
// interface.i
typedef double (*OP_FUNC)(double,double);
...
const OP_FUNC ADD = op_add;
const OP_FUNC SUB = op_sub;
const OP_FUNC MUL = op_mul;
...
```

When wrapped, this would create the constants ADD,SUB, and MUL containing the addresses of C callback functions. We could then pass these to other C functions expecting such function pointers as arguments as shown (for Tcl) :

```
%do_operation $ADD 3 4
7
%
```

Structures, unions, and object oriented C programming

If SWIG encounters the definition of a structure or union, it will create a set of accessor functions for you. While 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 mapped into the following set of accessor functions :

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

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 will assume that the name of the object is ‘Vector’ and create accessor functions like before. If two different names are used like this :

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

the name ‘Vector’ will still be used instead of “vector_struct”.

Character strings and structures

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

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

This results in the following accessor functions :

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

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

This seems to work most of the time, but occasionally it’s not always what you want. Typemaps can be used to change this behavior if necessary.

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 array itself. This restriction is due to the fact that C won’t let us change the “value” of an array. When this situation is detected, SWIG generates a warning message such as the following :

```
interface.i : Line 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 (and best reserved for experienced users). Otherwise, if you get this warning, it may be harmless.

C constructors and destructors

While not part of the C language, it is usually useful to have some mechanism for creating and destroying an object. You can, of course, do this yourself by making an appropriate call to `malloc()`, but SWIG can make such functions for you automatically if you use C++ syntax like this :

```
%module mymodule
...
struct Vector {
       Vector();            // Tell SWIG to create a C constructor
       ~Vector();           // Tell SWIG to create a C destructor
       double x,y,z;
}
```

When used with C code, SWIG will create two additional functions like this :

```
Vector *new_Vector() {
       return (Vector *) malloc(sizeof(Vector));
}

void delete_Vector(Vector *v) {
       free(v);
}
```

While C knows nothing about constructors and destructors, SWIG does---and it can automatically create some for you if you want. This only applies to C code--handling of C++ is handled differently.

As an alternative to explicitly defining constructors and destructors, SWIG can also automatically generate them using either a command line option or a pragma. For example :

```
swig -make_default example.i
```

or

```
%module foo
...
%pragma make_default                 // Make default constructors
... declarations ...
%pragma no_default                   // Disable default constructors
```

This works with both C and C++.

Adding member functions to C structures

Many scripting 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 typically operate on the structure (or object) in some way or another. While 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 scripting language 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 doing the following in an interface file :

```
// 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 in the definition of the Vector structure. For example:

```
// 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 {
              double magnitude();   // This will call Vector_magnitude
              ...
       }
} Vector;
```

So why bother with all of this `%addmethods` business? In short, you can use it to make some pretty cool ‘object oriented’ scripting language interfaces to C programs without having to rewrite anything in C++.

Nested structures

Occasionally, a C program will involve structures like this :

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

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

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

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

SWIG will 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 ...
```

Is it hairy? You bet. Does it work? Well, surprisingly yes. When used with Python and Perl5 shadow classes, it's even possible to access the nested members just like you expect :

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

If you've got a bunch of nested structure declarations, it is certainly advisable to check them out after running SWIG. However, there is a good chance that they will work. If not, you can always remove the nested structure declaration and write your own set of accessor functions.

Other things to note about structures

SWIG doesn't care if the definition of a structure 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 simply passing pointers around, this can be done without ever giving SWIG a structure definition.

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

C++ support

SWIG's support for C++ is an extension of the support for C functions, variables, and structures. However, SWIG only supports a subset of the C++ language. It has never been my goal to write a full C++ compiler or to turn scripting languages into some sort of weird pseudo C++ interpreter (considering how hard it is to write a C++ compiler, I'm not sure this would even be feasible anyways).

This section describes SWIG's low-level access to C++ declarations. In many instances, this low-level interface may be hidden by shadow classes or an alternative calling mechanism (this is usually language dependent and is described in detail in later chapters).

Supported C++ features

SWIG supports the following C++ features :

- Simple class definitions
- Constructors and destructors
- Virtual functions
- Public inheritance (including multiple inheritance)
- Static functions
- References

The following C++ features are not currently supported :

- Operator overloading
- Function overloading (without renaming)
- Templates (anything that would be defined using the '`template`' keyword).
- Friends
- Nested classes
- Namespaces
- Pointers to member functions.

Since SWIG's C++ support is a "work in progress", many of these limitations may be lifted in future releases. In particular, function overloading and nested classes, may be supported in the future. Operator overloading and templates are unlikely to be supported anytime in the near future, but I'm not going to rule out the possibility in later releases.

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 are handled with respect to differences between C and C++. It also enables the detection of C++ keywords. Without the `-c++` flag, SWIG will either issue a warning or a large number of syntax errors if it encounters any C++ code in an interface file.

Constructors and destructors

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

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

If the original C++ class does not have any constructors or destructors, putting constructors and destructors in the SWIG interface file will cause SWIG to generate wrappers for the default constructor and destructor of an object.

Member functions

Member functions are translated into accessor functions like this :

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

Virtual member functions are treated in an identical manner since the C++ compiler takes care of this for us automatically.

Static members

Static member functions are called directly without making any additional C wrappers. For example, the static member function `print(List  *l)` will simply be called as `List::print(List *l)` in the resulting wrapper code.

Member data

Member data is handled in exactly the same manner as used 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
        ...
        };
```

Protection

SWIG can only wrap class members that are declared public. Anything specified in a private or protected section will simply be ignored. To simplify your interface file, you may want to consider eliminating all private and protected declarations (if you've copied a C++ header file for example).

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 in a similar manner.

References

C++ references are supported, but SWIG will treat them as pointers. For example, a declaration like this :

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

will be accessed using a function like this :

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

Functions returning a reference will be mapped into functions returning pointers.

Inheritance

SWIG supports basic 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 will allow 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 will be parsed, but ignored).

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:
       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 Perl5, we can now perform the following operations :

```
beazley@slack% perl5.003
use shapes;
$circle = shapes::new_Circle(7);
$square = shapes::new_Square(10);
print shapes::Circle_area($circle),”\n”;
# Notice use of base class below
print shapes::Shape_area($circle),”\n”;
print shapes::Shape_area($square),"\n";
shapes::Shape_set_location($square,2,-3);
print shapes::Shape_perimeter($square),"\n";
<ctrl-d>
153.93804004599999757
153.93804004599999757
100.00000000000000000
40.00000000000000000
```

In our example, we have created Circle and Square objects. We can call member functions on each object by making calls to `Circle_area`, `Square_area`, and so on. However, we can can accomplish the same thing by simply using the `Shape_area` function on either object.

Templates

SWIG does not support template definitions--that is, it does not support anything that would be declared in C++ using the '`template`' keyword. If a template definition is found, SWIG will issue a warning message and attempt to ignore the contents of the entire declaration. For example, a template class such as the following would be ignored by SWIG :

```
// File : list.h
#define MAXITEMS 100
template<class T> class List {              // Entire class is ignored by SWIG
private:
    T *data;
    int nitems;
public:
    List() {
      data = new T [MAXITEMS];
      nitems = 0;
    }
    ~List() {
      delete [] data;
    };
    void append(T obj) {
      if (nitems < MAXITEMS) {
        data[nitems++] = obj;
      }
    }
    int length() {
      return nitems;
    }
    T get(int n) {
      return data[n];
    }
};
```

However, SWIG can support instantiations of a template and types involving templates. For example, the following interface file would be legal :

```
// SWIG interface involving a template
%module example
%{
#include "list.h"             // Get Template definition
%}

// Now a function involving templates
```

```
extern void PrintData(List<double> &l);
```

The type "`List<double>`" becomes the datatype for the function parameter. In, Python it might appear like this :

```
>>> print cl
_80a2df8_List<double>_p
>>>
```

To create specific objects, you may need to supply helper functions such as the following :

```
%inline %{
// Helper function to create a List<double>
List<double> *new_DoubleList() {
       return new List<double>;
}
%}
```

Specific templates can also be wrapped in a clever way using `typedef.` For example, the following would also work :

```
%module example
%{
#include "list.h"
typedef List<double> DoubleList;
%}

class DoubleList {
public:
       DoubleList();
       ~DoubleList();
       void append(double);
       int length();
       double get(int n);
};
```

In this case, SWIG thinks that there is a class "`DoubleList`" with the methods supplied. It generates the appropriate code and everything works like you would expect (of course, in reality there is no such class). When the SWIG module is compiled, all of the methods get supplied by the original template class. A key thing to keep in mind when working with templates is that SWIG can only handle particular instantiations of a template (such as a list of double). More general support is not yet provided (but may be added someday).

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 be changed by specifying

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

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.

Partial class definitions

Since SWIG is still somewhat limited in its support of C++, it may be necessary to only use partial class information in an interface file. This should not present a problem as SWIG does not need the exact C++ specification. As a general rule, you should strip all classes of operator overloading, friends, and other declarations before giving them to SWIG (although SWIG will generate only warnings for most of these things).

As a rule of thumb, running SWIG on raw C++ header or source files is currently discouraged. Given the complexity of C++ parsing and limitations in SWIG's parser it will still take some time for SWIG's parser to evolve to a point of being able to safely handle most raw C++ files.

SWIG, C++, and the Legislation of Morality

As languages go, C++ is quite possibly one of the most immense and complicated languages ever devised. It is certainly a far cry from the somewhat minimalistic nature of C. Many parts of C++ are designed to build large programs that are "safe" and "reliable." However, as a result, it is possible for developers to overload operators, implement smart pointers, and do all sorts of other insane things (like expression templates). As far as SWIG is concerned, the primary goal is attaching to such systems and providing a scripting language interface. There are many features of C++ that I would not have the slightest idea how to support in SWIG (most kinds of templates for example). There are other C++ idioms that may be unsafe to use with SWIG. For example, if one implements "smart" pointers, how would they actually interact with the pointer mechanism used by SWIG?

Needless to say, handling all of the possible cases is probably impossible. SWIG is certainly not guaranteed to work with every conceivable type of C++ program (especially those that use C++ in a maximal manner). Nor is SWIG claiming to build C++ interfaces in a completely "safe" manner. The bottom line is that effective use of C++ with SWIG requires that you know what you're doing and that you have a certain level of "moral flexibility" when it comes to the issue of building a useful scripting language interface.

The future of C++ and SWIG

SWIG's support of C++ is best described as an ongoing project. It will probably remain evolutionary in nature for the foreseeable future. In the short term, work is already underway for supporting nested classes and function overloading. As always, these developments will take time. Feedback and contributions are always welcome.

Objective-C

One of SWIG's most recent additions is support for Objective-C parsing. This is currently an experimental feature that may be improved in future releases.

Objective-C support is built using the same approach as used for C++ parsing. Objective-C interface definitions are converted into a collection of ANSI C accessor functions. These accessor functions are then wrapped by SWIG and turned into an interface.

To enable Objective-C parsing, SWIG should be given the `-objc` option (this option may be used in conjunction with the `-c++` option if using Objective-C++). It may also be helpful to use the -o option to give the output file the .m suffix needed by many Objective-C compilers. For example :

```
% swig -objc -o example_wrap.m example.i
```

Objective-C interfaces should also include the file ‘`objc.i`’ as this contains important definitions that are common to most Objective-C programs.

Objective-C Example

The following code shows a SWIG interface file for a simple Objective-C class :

```
%module list
%{
#import “list.h”
%}
%include objc.i
// Very simple list class
@interface List : Object {
     int   nitems;                 // Number of items in the list
     int   maxitems;               // Maximum number of items
     id   *items;                  // Array holding the items
}
//-------------------------  List methods -------------------------

// Create a new list
+ new;

// Destroy the list
- free;

// Copy a list
- copy;

// Append a new item to the list
- (void) append: (id) item;

// Insert an item in the list
- (void) insert: (id) item : (int) pos;

// Delete an item from the list
-  remove: (int) pos;

// Get an item from the list
- get: (int) i;

// Find an item in the list and return its index
- (int) index: obj;

// Get length of the list
- (int) len;

// Print out a list (Class method)
+ (void) print: (List *) l;

@end
```

Constructors and destructors

By default, SWIG assumes that the methods “new” and “free” correspond to constructors and destructors. These functions are translated into the following accessor functions :

```
List *new_List(void) {
       return (List *) [List new];
}
void delete_List(List *l) {
       [l free];
}
```

If the original Objective-C class does not have any constructors or destructors, putting them in the interface file will cause SWIG to generate wrappers for a default constructor and destructor (assumed to be defined in the object’s base-class).

If your Objective-C program uses a different naming scheme for constructors and destructors, you can tell SWIG about it using the following directive :

```
%pragma objc_new = “create”         // Change constructor to ‘create’
%pragma objc_delete = “destroy”     // Change destructor to ‘destroy’
```

Instance methods

Instance methods are converted into accessor functions like this :

```
void List_append(List *l, id item) {
       [l append : item];
}
```

Class methods

Class methods are translated into the following access function :

```
void List_print(List *l) {
       [List print : l];
}
```

Member data

Member data is handled in the same manner as for C++ with accessor functions being produced for getting and setting the value. By default, all data members of an Objective-C object are private unless explicitly declared `@public`.

Protection

SWIG can only wrap data members that are declared @public. Other protection levels are ignored.

The use of id

The datatype ‘`id`’ assumes the same role in SWIG as it does with Objective-C. A function operating on an ‘id’ will accept any Object type (works kind of like `void *`). All methods with no explicit return value are also assumed to return an ‘id’.

Inheritance

Essentially all Objective-C classes will inherit from a baseclass `Object`. If undefined, SWIG will generate a warning, but other function properly. A missing baseclass has no effect on the wrapper code or the operation of the resulting module. Really, the only side effect of a missing base class is that you will not be able to execute base-class methods from the scripting interface. Objective-C does not support multiple inheritance.

Referring to other classes

The `@class` declaration can be used to refer to other classes. SWIG uses this in certain instances to make sure wrapper code is generated correctly.

Categories

Categories provide a mechanism for adding new methods to existing Objective-C classes. SWIG correctly parses categories and attaches the methods to the wrappers created for the original class. For example :

```
%module example
%{
#import “foo.h”
%}

// Sample use of a category in an interface
@interface Foo (CategoryName)
// Method declarations
-bar : (id) i;
@end
```

Implementations and Protocols

SWIG currently ignores all declarations found inside an `@implementation` or `@protocol` section. Support for this may be added in future releases.

Although SWIG ignores protocols, protocol type-specifiers may be used. For example, these are legal declarations :

```
%module example

%interface Foo : Object <proto1, proto2> {

}
// Methods
- Bar : (id <proto1,proto2>) i;
@end
```

SWIG will carry the protocol lists through the code generation process so the resulting wrapper code compiles without warnings.

Renaming

Objective-C members can be renamed using the `%name()` directive as in :

```
@interface List : Object {
```

```
@public
%name(size) int length;                    // Rename length to size
}

+ new;
- free;
%name(add) -(void) append: (id) item;      // Rename append to add
@end
```

Adding new methods

New methods can be added to a class using the `%addmethods` directive. This is primarily used with shadow classes to add additional functionality to a class. For example :

```
@interface List : Object {
}
... bunch of Objective-C methods ...
%addmethods {
        - (void) output {
                ... code to output a list ...
        }
}
@end
```

`%addmethods` works in exactly the same manner as it does for C and C++ (except that Objective-C syntax is allowed). Consult those sections for more details.

Other issues

Objective-C is dynamically typed while SWIG tends to enforce a type-checking model on all of its pointer values. This mismatch could create operational problems with Objective-C code in the form of SWIG type errors. One solution to this (although perhaps not a good one) is to use the SWIG pointer library in conjunction with Objective-C. The pointer library provides simple functions for casting pointer types and manipulating values.

Certain aspects of Objective-C are not currently supported (protocols for instance). These limitations may be lifted in future releases.

Conditional compilation

SWIG does not run the C preprocessor, but it does support conditional compilation of interface files in a manner similar to the C preprocessor. This can be done by placed `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif`, and `#endif` directives in your interface file. These directives can be safely nested. This allows one to conditionally compile out troublesome C/C++ code if necessary. For example, the following file can serve as both a C header file and a SWIG interface file :

```
#ifdef SWIG
%module mymodule
%{
#include “header.h”
%}

%include wish.i
```

```
#endif

... normal C declarations here ...
```

Similarly, conditional compilation can be used to customize an interface. The following interface file can be used to build a Perl5 module that works with either static or dynamic linking :

```
%module mymodule
%{
#include "header.h"
%}

... Declarations ...

#ifdef STATIC
%include perlmain.i          // Include code for static linking
#endif
```

However, it is not safe to use conditional compilation in the middle of a declaration. For example :

```
double foo(
#ifdef ANSI_ARGS
double a, double b
#endif
);
```

This fails because the SWIG parser is not equipped to handle conditional compilation directives in an arbitrary location (like the C preprocessor). For files that make heavy use of the C preprocessor like this, it may be better to run the header file through the C preprocessor and use the output as the input to SWIG.

Defining symbols

To define symbols, you can use the `-D` option as in :

```
swig -perl5 -static -DSTATIC interface.i
```

Symbols can also be defined using `#define` with no arguments. For example :

```
%module mymodule
#define STATIC

... etc ...
```

For the purposes of conditional compilation, one should not assign values to symbols. If this is done, SWIG interprets the #define as providing the definition of a scripting language constant.

The #if directive

The `#if` directive can only be used in the following context :

```
#if defined(SYMBOL)
...
```

```
#elif !defined(OTHERSYMBOL)
...
#endif
```

The C preprocessor version supports any constant integral expression as an argument to `#if`, but SWIG does not yet contain an expression evaluator so this is not currently supported. As a result, declarations such as the following don't yet work :

```
#if (defined(foo) || defined(bar))
...
#endif
```

Predefined Symbols

One or more of the following symbols will be defined by SWIG when it is processing an interface file :

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
SWIGPERL4	Defined when using Perl4
SWIGPERL5	Defined when using Perl5
SWIGPYTHON	Defined when using Python
SWIGGUILE	Defined when using Guile
SWIGWIN	Defined when running SWIG under Windows
SWIGMAC	Defined when running SWIG on the Macintosh

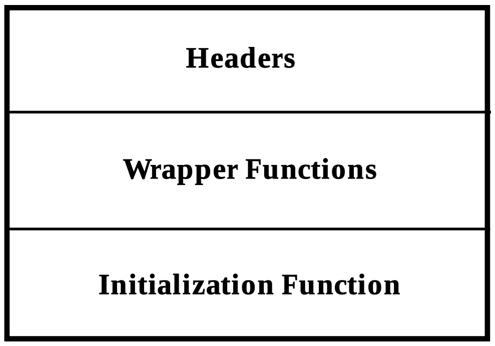
Interface files can look at these symbols as necessary to change the way in which an interface is generated or to mix SWIG directives with C code. These symbols are also defined within the C code generated by SWIG (except for the symbol '`SWIG`' which is only defined within the SWIG compiler).

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 ways to insert code, but it's useful to know how the output of SWIG is structured first.

The output of SWIG

SWIG creates a single C source file containing wrapper functions, initialization code, and support code. The structure of this file is as follows :



The headers portion typically contains header files, supporting code, helper functions, and forward declarations. If you look at it, you'll usually find a hideous mess since this also contains the SWIG run-time pointer type-checker and internal functions used by the wrapper functions. The "wrapper" portion of the output contains all of the wrapper functions. Finally, the initialization function is a single C function that is created to initialize your module when it is loaded.

Code blocks

A code block is enclosed by a `%{, %}` and is used to insert code into the header portion of the resulting wrapper file. Everything in the block is copied verbatim into the output file and will appear before any generated wrapper functions. Most SWIG input files have at least one code block that is normally used to include header files and supporting C code. Additional code blocks may be placed anywhere in a SWIG file as needed.

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

... Declare functions here
%{

... Include Tcl_AppInit() function here ...

%}
```

Code blocks are also typically used to write "helper" functions. These are functions that are used specifically for the purpose of building an interface and 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

Because 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 fed into the SWIG parser and turned into an interface. 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 the `%{ ... %}` block.

Initialization blocks

Code may also be inserted using an initialization block, as shown below :

```
%init %{

       init_variables();
%}
```

This code is inserted directly into SWIG's initialization function. You can use it to perform additional initialization and operations. Since this code is inserted directly into another function, it should not declare functions or include header files. Primarily this can be used to add callouts to widgets and other packages that might also need to be initialized when your extension is loaded.

Wrapper code blocks

Code may be inserted in the wrapper code section of an output file using the `%wrapper` directive as shown :

```
%wrapper %{
       ... a bunch of code ...
%}
```

This directive, for almost all practical purposes, is identical to just using a `%{,%}` block, but may be required for more sophisticated applications. It is mainly only used for advanced features in the SWIG library. As a general rule, you should avoid using this directive unless you absolutely know what you are doing.

A general 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. Either way, this step is fairly easy.
- Make sure everything in the interface file uses ANSI C / C++syntax.
- Check to make sure there aren't any functions involving function pointers, or variable length arguments since SWIG doesn't like these very much.
- Eliminate unnecessary C preprocessor directives. SWIG will probably remove most of them, but better safe than sorry. Remember, SWIG does not run the C preprocessor.
- 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.

While this may sound complicated, the process turns out to be relatively easy in practice--for example, making an interface to the entire OpenGL library only takes about 5-10 minutes.

In the process of building an interface, you are encouraged to use SWIG to find problematic declarations and specifications. SWIG will report syntax errors and other problems along with the associated file and line number.

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?

While 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 for doing 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. Therfore, 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 structure and organization. For example , you can break the interface up into sections, provide documentation, and do other things that you might not normally do with an ordinary .h file.
- 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 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. You can have SWIG include certain header files by using a `%{,%}` block as follows :

```
%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 must be 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. This is the brute force approach.
- 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.

Working with the C preprocessor

If you have a header file that makes heavy use of macros and C preprocessor directives, it may be useful to run it through the C preprocessor first. This can usually be done by running the C compiler with the -E option. The output will be completely hideous, but macros and other preprocessor directives should now be expanded as needed. If you want to wrap a C preprocessor macro with SWIG, this can be done by giving a function declaration with the same name and usage as the macro. When writing the macro as a function declaration, you are providing SWIG with type-information--without that, SWIG would be unable to produce any sort of wrapper code.

How to cope with C++

Given the complexity of C++, it will almost always be necessary to build a special interface file containing suitably edited C++ declarations. If you are working with a system involving 400 header files, this process will not be trivial. Perhaps the best word of advice is to think hard about what you want this interface to be. Also, is it absolutely critical to wrap every single function in a C++ program? SWIG's support of C++ will improve with time, but I'll be the first to admit that SWIG works much better with pure ANSI C code when it comes to large packages.

How to avoid creating the interface from hell

SWIG makes it pretty 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 chunk of code being produced in the resulting wrapper file. To give you an idea, 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. I can only imagine what wrapping a huge C++ class hierarchy would generate. Here's a few rules of thumb for making smaller interfaces :

- Ask yourself if you really need to access particular functions. It is usually not necessary to wrap every single function in a package. In fact, you probably only need a relatively small subset.
- 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 the problem at hand. 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.

Writing a nice interface to a package requires work. Just because you use SWIG it doesn't mean that you're going to end up with a good interface. SWIG is primarily designed to eliminate the tedious task of writing wrapper functions. It does not eliminate the need for proper planning and design when it comes to building a useful application. In short, a little forethought can go a long way.

Of course,if you're primarily interested in just slapping something together for the purpose of debugging, rapid application development, and prototyping, SWIG will gladly do it for you (in fact, I use SWIG alot for this when developing other C/C++ applications).

Multiple files and the SWIG library 4

For increased modularity and convenience, it is usually useful to break an interface specification up into multiple files or modules. SWIG provides a number of features for doing just this.

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”
```

When used in this manner, SWIG will create a single wrapper file containing all of the included functions. [1]

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 will automatically declare all functions it finds as “extern”. Thus, use of a header file may not be required in this case.

The %extern directive

The `%extern` directive is like `%include` except that it only scans a file for type and class information. It does not actually wrap anything found in the input file. This directive is primarily used for handling class hierarchies and multiple modules. For example :

```
%module derived
%extern baseclass.h         // Grab definition of a base class

// Now wrap a derived class
```

1. If you are using dynamic loading, this may be unncessary as each module can be wrapped individually and loaded into the scripting language.

```
class Derived : public BaseClass {
public:
        ...
};
```

This interface file would grab the member functions and data from a baseclass, but only use them in the specification of a derived class. `%extern` processing of files is also useful for picking up common typedefs and definitions in a large package.

The %import directive

The `%extern` directive is used to gather declarations from files that you don't want to wrap into an interface. Unfornunately, the exact role of these files is not always clear. They could just contain definitions, or they might correspond to an entrirely different SWIG module. The `%import` directive is a stronger version of `%extern` that tells SWIG that all of the declarations in the file are indeed, in an entirely different module. This information may affect the code generated by various language modules since they will have a better idea of where things are defined and how they are to be used.

Including files on the command line

Much 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 approach turns out to be particularly useful for debugging and building extensions to different kinds of languages. When libraries are specified in this manner, they are included <u>after</u> 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. While 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`, SWIG will search for the file 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, you can 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 *);
```

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

While 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, we 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 cool Perl script that you liked to use alot. 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
```

Support files can also be checked out within an interface file using the `%checkout` directive.

```
%checkout myscript.pl
```

This will attempt to copy the file `myscript.pl` to the current directory when SWIG is run. If the file already exists, nothing is done.

The world's fastest way to write a Makefile

Since the SWIG library is not restricted to SWIG interface files, it can be used to hold other kinds of useful files. For example, if you need a quick Makefile for building Tcl extensions, 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.

Checking in library files

It is also possible to check files into the SWIG library. If you've made an interesting interface that you would like to keep around, simply type :

```
unix > swig -ci -python cool.i
```

and the file ‘`cool.i`’ will be placed in the Python section of the library. If your interface file is general purpose, you can install it into the general library as follows :

```
unix > swig -ci ../cool.i
```

When files are checked in, they are placed into the directory defined by the `SWIG_LIB` variable that was used during SWIG compilation or the `SWIG_LIB` environment variable (if set). If you don’t know what the value of this variable is, type the following to display its location.

```
unix > swig -swiglib
/usr/local/lib/swig_lib
unix >
```

In order to check files into the library, you must have write permission on the library directory. For this reason, one of the primary uses for the `-ci` option is to provide a simple mechanism for installing SWIG extensions. If these extensions need to install library files, that can be done by simply running SWIG.

Static initialization of multiple modules

When using static linking, some language modules allow multiple modules to be initialized as follows :

```
%module package, equations, graphics, fileio, data, network, user

... More declarations ...
```

The module list can contain SWIG generated modules or third-party applications. Refer to the appropriate language chapter for a detailed description of this feature.

More about the SWIG library

Full documentation about the SWIG library is included in the SWIG source distribution. In fact, the documentation is automatically generated by SWIG, which leads us to the next section...

Documentation System 5

Introduction

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

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

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

How it works

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

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

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

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

The first line shows how to call the function, the second line shows some additional information about the function (related to its C implementation), while the third line contains the comment text. The first two lines are automatically generated by SWIG and may be different for each language module. For example, the Perl5 module would generate the following output :

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

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

Choosing a documentation format

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

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

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

Function usage and argument names

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

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

will produce the following documentation (for Python) :

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

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

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

Titles, sections, and subsections

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

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

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

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

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

%section “Mathematical Functions”

... declarations ...

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

%section “File I/O”

... Declarations ...
```

Formatting

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

```
%title “My Documentation”, sort, before, pre
```

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

If formatting information is specified for a section like this

```
%subsection “3D Graphics”, nosort, after
```

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

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

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

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

Default Formatting

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

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

Comment Formatting variables

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

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

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

Sorting

Documentation can be sorted using the '`sort`' parameter. For example :

```
%title "My interface",sort
```

When used, all documentation entries, including sections will be alphabetically sorted. Sorting can be disabled in particular sections and subsection by specifying the '`nosort`' parameter in a section declaration. By default, SWIG does not sort documentation. As a general rule, it really only comes in handy if you have a really messy interface file.

For backwards compatibility with earlier versions of SWIG, the following directives can be used to specify sorting.

```
%alpha        Sort documentation alphabetically (obsolete)
%raw          Keep documentation in order (obsolete)
```

These directives only apply globally and should near the beginning of file. Future support of these directives is not guaranteed and generally discouraged.

Comment placement and formatting

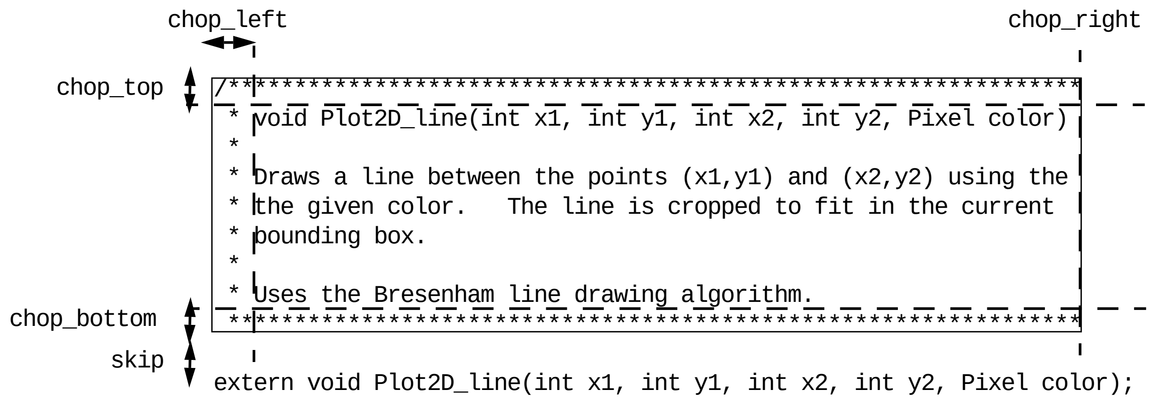
Comments may be placed before or after a declaration. This is specified using the '`before`' and '`after`' parameters. The space between a comment and a declaration can be set by changing the '`skip`' parameter. By default, `skip=1`, indicating that a comment and declaration must be on adjacent lines. Use of the `skip` parameter makes it possible for the documentation generator to ignore comments that are too far away and possibly unrelated to a declaration.

By default, SWIG reformats the text found in a comment. However, in many cases, your file may have preformatted comments or comment blocks. To handle such comments correctly, you can use preformatted mode. This is specified using the 'pre' parameter as follows :

```
%section "Preformatted Section",pre
%section "Reformatted Section", format
```

All declarations in this section will now be assumed to have preformatted comments. When using the preformat mode, a variety of other parameters are available as shown in the following diagram :

.



```
/*****************************************************************
 * void Plot2D_line(int x1, int y1, int x2, int y2, Pixel color)
 *
 * Draws a line between the points (x1,y1) and (x2,y2) using the
 * the given color.   The line is cropped to fit in the current
 * bounding box.
 *
 * Uses the Bresenham line drawing algorithm.
 *****************************************************************/
extern void Plot2D_line(int x1, int y1, int x2, int y2, Pixel color);
```

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

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

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

       Uses the Bresenham line drawing algorithm.
```

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

Tabs and other annoyances

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

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

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

Ignoring comments

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

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

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

C Information

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

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

Adding Additional Text

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

```
%text %{

This is some additional documentation text.

%}
```

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

Disabling all documentation

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

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

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

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

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

An Example

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

```
//
// array.i
// This SWIG library file provides access to C arrays.

%module carray

%section "SWIG C Array Module",info,after,pre,nosort,skip=1,chop_left=3,
chop_right=0,chop_top=0,chop_bottom=0

%text %{
%include array.i
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

%subsection "String Arrays"

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

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

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

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

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

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

```
#elif defined(SWIGPYTHON)

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

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

#endif

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

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

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

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

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

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

```
7.  SWIG C Array Module
=======================

%include array.i
```

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

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

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

7.1.  Integer Arrays
--------------------
The following functions provide access to integer arrays (mapped
onto the C 'int' datatype.


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

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

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

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

7.2.  Floating Point Arrays
---------------------------
The following functions provide access to arrays of floats and doubles.


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

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

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

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

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

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

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

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

7.3.  String Arrays
-------------------

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

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

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

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

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

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

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

ASCII Documentation

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

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

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

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

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

HTML Documentation

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

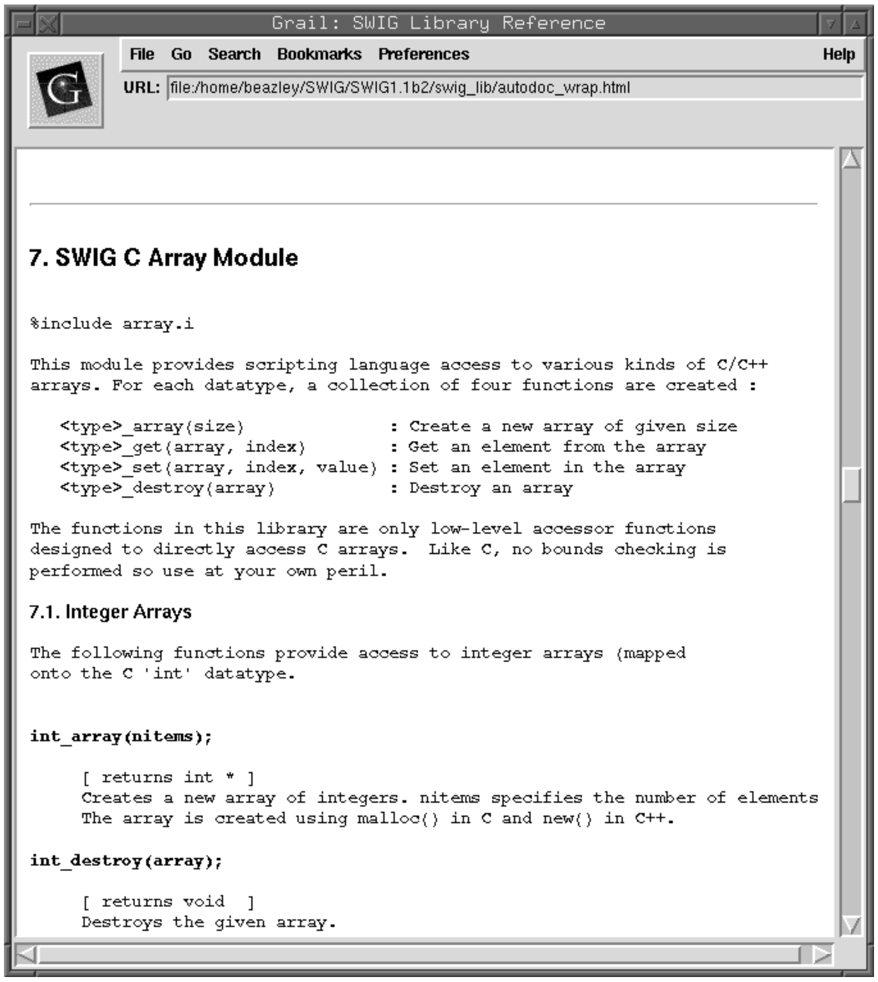
```
html_title = "<H1>:</H1>"
html_contents = "<H1>:</H1>"
html_section = "<HR><H2>:</H2>"
html_subsection = "<H3>:</H3>"
html_subsubsection = "<H4>:</H4>"
html_usage = "<B><TT>:</TT></B>"
html_descrip = "<BLOCKQUOTE>:</BLOCKQUOTE>"
html_text = "<P>"
html_cinfo = ""
html_preformat = "<PRE>:</PRE>"
html_body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

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

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

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

Sample HTML output is shown below :


Grail: SWIG Library Reference
File Go Search Bookmarks Preferences Help
URL: file:/home/beazley/SWIG/SWIG1.1b2/swig_lib/autodoc_wrap.html

7. SWIG C Array Module

```
%include array.i

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

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

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

7.1. Integer Arrays

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

int_array(nitems);

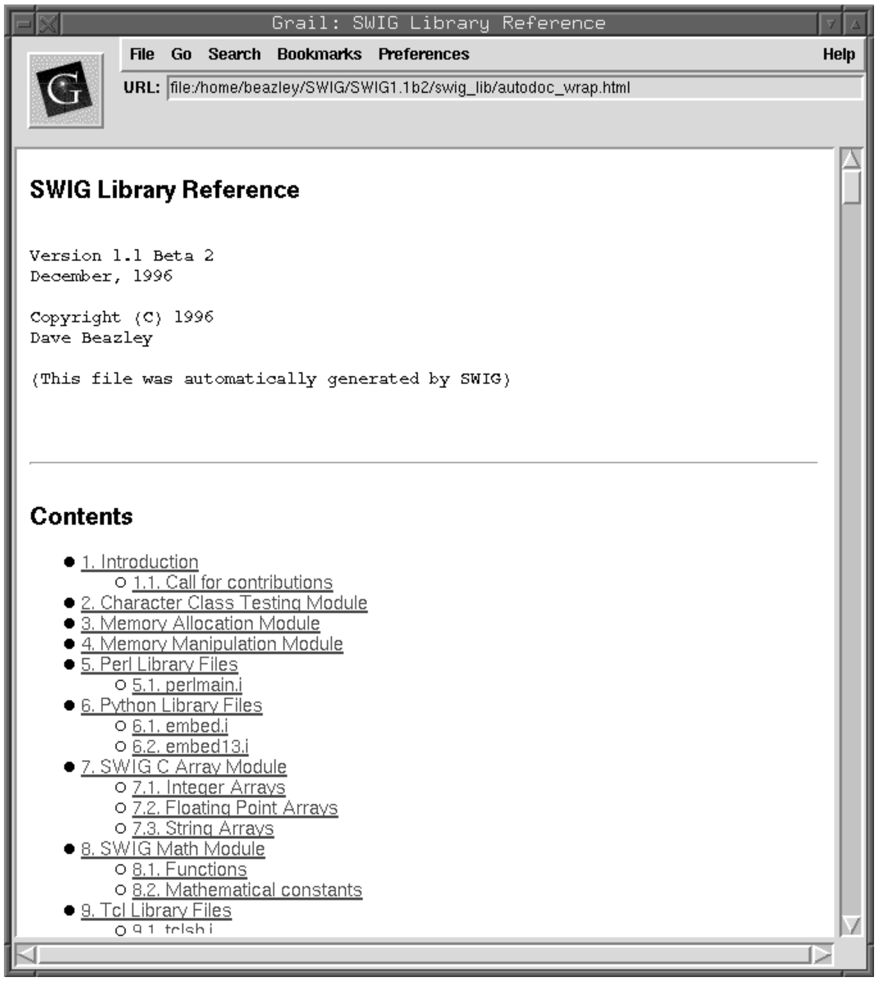
```
     [ returns int * ]
     Creates a new array of integers. nitems specifies the number of elements
     The array is created using malloc() in C and new() in C++.
```

int_destroy(array);

```
     [ returns void  ]
     Destroys the given array.
```

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

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


Grail: SWIG Library Reference
File Go Search Bookmarks Preferences Help
URL: file:/home/beazley/SWIG/SWIG1.1b2/swig_lib/autodoc_wrap.html

SWIG Library Reference

Version 1.1 Beta 2
December, 1996

Copyright (C) 1996
Dave Beazley

(This file was automatically generated by SWIG)

Contents

- 1. Introduction
  - 1.1. Call for contributions
- 2. Character Class Testing Module
- 3. Memory Allocation Module
- 4. Memory Manipulation Module
- 5. Perl Library Files
  - 5.1. perlmain.i
- 6. Python Library Files
  - 6.1. embed.i
  - 6.2. embed13.i
- 7. SWIG C Array Module
  - 7.1. Integer Arrays
  - 7.2. Floating Point Arrays
  - 7.3. String Arrays
- 8. SWIG Math Module
  - 8.1. Functions
  - 8.2. Mathematical constants
- 9. Tcl Library Files
  - 9.1. tclsh.i

LaTeX Documentation

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

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

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

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

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

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

C++ Support

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

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

The Final Word?

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

Pointers, Constraints, and Typemaps 6

Introduction

For most applications, SWIG's treatment of basic datatypes and pointers is enough to build an interface. However, in certain cases, it is desirable to change SWIG's treatment of particular datatypes. For example, we may want a `char **` to act like a list of strings instead of a pointer. In another instance, we may want to tell SWIG that `double *result` is the output value of a function. Similarly, we might want to map a datatype of `float[4]` into a 4 element tuple. This chapter describes advanced methods for managing pointers, arrays, and complex datatypes. It also describes how you can customize SWIG to handle new kinds of objects and datatypes.

The SWIG Pointer Library

If your interface involves C pointers, chances are you will need to work with these pointers in some way or another. The SWIG pointer library provides a collection of useful methods for manipulating pointers. To use the library, simply put the following declaration in your interface file :

```
%include pointer.i          // Grab the SWIG pointer library
```

or run SWIG as follows :

```
swig -perl5 -lpointer.i interface.i
```

Doing so adds a collection of pointer manipulation functions that are described below. The functions are mainly designed to work with basic C datatypes, but can often be used with more complicated structures.

Pointer Library Functions

`ptrcreate(type,?value?,?nitems?)`

Creates a new object and returns a pointer to it. `type` is a string containing the C datatype and may be one of "`int`","`short`","`long`","`float`","`double`","`char`","`char *`", or "`void`". `value` is the optional initial value to be assigned to the object. `nitems` is an optional parameter containing the number of objects to create. By default it is 1, but specifying another value allows you to create an array of values. This function is really just a wrapper around the C `malloc()` function.

ptrfree(ptr)

Destroys an object created by `ptrcreate`. It is generally unsafe to use this function on objects not created by `ptrcreate`. Calls the C `free()` function.

ptrvalue(ptr,?index?,?type?)

This dereferences a pointer and returns the value that it is pointing to. `index` is an optional parameter that allows array access by returning the value of `ptr[index]`. `type` is an optional parameter that explicitly specifies the datatype. Since SWIG pointers are encoded with type information, the type is usually unnecessary. The `type` parameter provides somewhat better performance and allows you to dereference a pointer of different type however.

ptrset(ptr, value, ?index?, ?type?)

Sets the value of the object a pointer is pointing to. `value` is the new value of the object. `index` is an optional parameter allowing array access by setting `ptr[index] = value`. `type` is an optional parameter that explicitly specifies the datatype as described above.

ptrcast(ptr, newtype)

Casts a pointer to a new datatype and returns the new value. `newtype` is a string containing the new datatype and may either be the "mangled" version used by SWIG (such as "`_Vector_p`") or the C version (such as "`Vector *`"). This function works with any kind of pointer value. In additional to pointers, `ptr` may also hold an integer value in which case the integer is turned into a pointer of given type.

ptradd(ptr, offset)

Adds an offset to a pointer and returns a new pointer. `offset` is specified as the number of objects except for unknown complex datatypes in which case it is the number of bytes. For example, is `ptr` is a "`double *`", `ptradd(ptr,1)` will return the next double. On the other hand, if if `ptr` is "`Vector *`", then `ptradd(ptr,1)` will update the pointer by 1 byte.

ptrmap(type1,type2)

This performs a "runtime typedef" and makes SWIG recognize pointers of `type1` and `type2` as equivalent. `type1` and `type2` are specified as strings. Not generally needed, but sometimes useful.

A simple example

Suppose you have the following C function :

```
void add(double a, double b, double *result) {
        *result = a + b;
}
```

To manage the result output, we can write an interface file like this :

```
%module example
%include pointer.i

extern void add(double a, double b, double *result);
```

Now, let's use the pointer library (shown for a few languages) :

```
# Tcl
set result [ptrcreate double]             ;# Create a double
add 4.5 3 $result                         ;# Call our C function
puts [ptrvalue $result]                   ;# Print out the result
ptrfree $result                           ;# Destroy the double

# Perl5
use example;
package example;                           # Functions are in example package
$result = ptrcreate("double");             # Create a double
add(4.5,3,$result);                        # Call C function
print ptrvalue($result),"\n";              # Print the result
ptrfree($result);                          # Destroy the double

# Python
import example
result = example.ptrcreate("double")      # Create a double
example.add(4.5,3,result)                 # Call C function
print example.ptrvalue(result)            # Print the result
example.ptrfree(result)                   # Destroy the double
```

In this case, the idea is simple--we create a pointer, pass it to our C function, and dereference it to get the result. It's essentially identical to how we would have done it in C (well, minus the function call to dereference the value).

Creating arrays

Now suppose you have a C function involving arrays :

```
void addv(double a[], double b[], double c[], int nitems) {
       int i;
       for (i = 0; i < nitems; i++) {
              c[i] = a[i]+b[i];
       }
}
```

This is also easily handled by our pointer library. For example (in Python) :

```
# Python function to turn a list into an "array"
def build_array(l):
       nitems = len(l)
       a = ptrcreate("double",0,nitems)
       i = 0
       for item in l:
              ptrset(a,item,i)
              i = i + 1
       return a

# Python function to turn an array into list
def build_list(a,nitems):
       l = []
       for i in range(0,nitems):
              l.append(ptrvalue(a,i))
       return l

# Now use our functions
a = listtoarray([0.0,-2.0,3.0,9.0])
```

```
b = build_array([-2.0,3.5,10.0,22.0])
c = ptrcreate("double",0,4)            # For return result
add(a,b,c,4)                           # Call our C function
result = build_list(c)                 # Build a python list from the result
print result
ptrfree(a)
ptrfree(b)
ptrfree(c)
```

This example may look quite inefficient on the surface (due to the translation of Python lists to and from C arrays). However, if you're working with lots of C functions, it's possible to simply pass C pointers around between them without any translation. As a result, applications can run fast--even when controlled from a scripting language. It's also worth emphasizing that the `ptrcreate()` function created a real C array that can be interchanged with other arrays. The `ptrvalue()` function can also dereference a C pointer even if it wasn't created from Python.

Packing a data structure

The pointer library can even be used to pack simple kinds of data-structures, perhaps for sending across a network, or simply for changing the value. For example, suppose you had this data structure:

```
struct Point {
       double x,y;
       int color;
};
```

You could write a Tcl function to set the fields of the structure as follows :

```
proc set_point { ptr x y c } {
       set p [ptrcast $ptr "double *"]          ;# Make a double *
       ptrset $p $x                             ;# Set x component
       set p [ptradd $p 1]                      ;# Update pointer
       ptrset $p $y                             ;# Set y component
       set p [ptrcast [ptradd $p 1] "int *"]    ;# Update pointer and cast
       ptrset $p $c                             ;# Set color component
}
```

This function could be used even if you didn't tell SWIG anything about the "Point" structure above.

Introduction to typemaps

Sometimes it's desirable to change SWIG behavior in some manner. For example, maybe you want to automatically translate C arrays to and from Perl lists. Or perhaps you would like a particular function argument to behave as an output parameter. Typemaps provide a mechanism for doing just this by modifying SWIG's code generator. Typemaps are new to SWIG 1.1, but it should be emphasized that they are not required to build an interface.

The idea (in a nutshell)

The idea behind typemaps is relatively simple--given the occurrence of a particular C datatype, we want to apply rules for special processing. For example, suppose we have a C function like this :

```
void add(double a, double b, double *result) {
       *result = a + b;
}
```

It is clear to us that the result of the function is being returned in the `result` parameter. Unfortunately, SWIG isn't this smart--after all "result" is just like any other pointer. However, with a typemap, we can make SWIG recognize "`double *result`" as a special datatype and change the handling to do exactly what we want.

So, despite being a common topic of discussion on the SWIG mailing list, a typemap is really just a special processing rule that is applied to a particular datatype. Each typemap relies on two essential attributes--a datatype and a name (which is optional). When trying to match parameters, SWIG looks at both attributes. Thus, special processing applied to a parameter of "`double *result`" will not be applied to "`double *input`". On the other hand, special processing defined for a datatype of "`double *`" could be applied to both (since it is more general).

Using some typemaps

It is easy to start using some typemaps right away. To wrap the above function, simply use the `typemaps.i` library file (which is part of the SWIG library) as follows :

```
// Simple example using typemaps
%module example
%include typemaps.i                 // Grab the standard typemap library

%apply double *OUTPUT { double *result };
extern void add(double a, double b, double *result);
```

The `%apply` directive tells SWIG that we are going to apply special processing to a datatype. The "`double *OUTPUT`" is the name of a rule describing how to return an output value from a "`double *`" (this rule is defined in the file `typemaps.i`). The rule gets applied to all of the datatypes listed in curly braces-- in this case "`double *result`".

While it may sound complicated, when you compile the module and use it, you get a function that works as follows :

```
# Perl code to call our add function

$a = add(3,4);
print $a,"\n";
7
```

Our function is much easier to use and it is no longer necessary to create a special double * object and pass it to the function. Typemaps took care of this automatically.

Managing input and output parameters

By default, when SWIG encounters a pointer, it makes no assumptions about what it is (well, other than the fact that it's a pointer). The `typemaps.i` library file contains a variety of methods for changing this behavior. The following methods are available in this file :

Input Methods

These methods tell SWIG that a pointer is a single input value. When used, functions will expect values instead of pointers.

```
int *INPUT
short *INPUT
long *INPUT
unsigned int *INPUT
unsigned short *INPUT
unsigned long *INPUT
double *INPUT
float *INPUT
```

Suppose you had a C function like this :

```
double add(double *a, double *b) {
       return *a+*b;
}
```

You could wrap it with SWIG as follows :

```
%module example
%include typemaps.i
...
extern double add(double *INPUT, double *INPUT);
```

Now, when you use your function ,it will work like this :

```
% set result [add 3 4]
% puts $result
7
```

Output Methods

These methods 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, but multiple return values can be 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. For example, suppose you had a function like this :

```
// Returns a status code and double
int get_double(char *str, double *result);
```

When declared in SWIG as :

```
int get_double(char *str, double *OUTPUT);
```

The function would return a list of output values as shown for Python below :as follows :

```
>>> get_double("3.1415926")         # Returns both a status and value
[0, 3.1415926]
>>>
```

Input/Output Methods

When a pointer serves as both an input and output value you can use the following methods :

```
int *BOTH
short *BOTH
long *BOTH
unsigned int *BOTH
unsigned short *BOTH
unsigned long *BOTH
double *BOTH
float *BOTH
```

A typical C function would be as follows :

```
void negate(double *x) {
       *x = -(*x);
}
```

To make x function as both and input and output value, declare the function like this in an interface file :

```
%module example
%include typemaps.i
...
extern void negate(double *BOTH);
```

Now within a script, you can simply call the function normally :

```
$a = negate(3);             # a = -3 after calling this
```

Using different names

By explicitly using the parameter names of INPUT, OUTPUT, and BOTH in your declarations, SWIG performs different operations. If you would like to use different names, you can simply use the `%apply` directive. 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 both
%apply long *BOTH {long *x};
```

`%apply` only renames the different type handling rules. You can use it to match up with the naming scheme used in a header file and so forth. To later clear a naming rule, the `%clear` directive can be used :

```
%clear double *result;
%clear Int32 *in, long *x;
```

Applying constraints to input values

In addition to changing the handling of various input values, it is also possible 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 (which is also based on typemaps).

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 built-in 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, we have only seen a high-level picture of typemaps and have utilized pre-existing typemaps in the SWIG library. However, it is possible to do more if you're willing to get your hands dirty and dig into the internals of SWIG and your favorite scripting language.

Before diving in, first ask yourself do I really need to change SWIG's default behavior? The basic pointer model works pretty well most of the time and I encourage you to use it--after all, I wanted SWIG to be easy enough to use so that you didn't need to worry about low level details. If, after contemplating this for awhile, you've decided that you really want to change something, a word of caution is in order. Writing a typemap from scratch usually requires a detailed knowledge of the internal workings of a particular scripting language. It is also quite easy to break all of the output code generated by SWIG if you don't know what you're doing. On the plus side, once a typemap has been written it can be reused over and over again by putting it in the SWIG library (as has already been demonstrated). This section describes the basics of typemaps. Language specific information (which can be quite technical) is contained in the later chapters.

Motivations for using typemaps

Suppose you have a few C functions such as the following :

```
void glLightfv(GLenum light, Glenum pname, GLfloat parms[4]);
```

In this case, the third argument takes a 4 element array. If you do nothing, SWIG will convert the last argument into a pointer. When used in the scripting language, you will need to pass a "`GLfloat *`" object to the function to make it work.

Managing special data-types with helper functions

Helper functions provide one mechanism for dealing with odd datatypes. With a helper function, you provide additional functionality for creating and destroying objects or converting values into a useful form. These functions are usually just placed into your interface file with the rest of the functions. For example, a few helper functions to work with 4 element arrays for the above function, might look like this :

```
%inline %{
/* Create a new GLfloat [4] object */
GLfloat *newfv4(double x, double y, double z, double w) {
       GLfloat *f = (GLfloat *) malloc(4*sizeof(GLfloat));
       f[0] = x;
       f[1] = y;
       f[2] = z;
       f[3] = w;
       return f;
}

/* Destroy a GLfloat [4] object */
void delete_fv4(GLfloat *d) {
       free(d);
}
%}
```

When wrapped, our helper functions will show up the interface and can be used as follows :

```
% set light [newfv4 0.0 0.0 0.0 1.0]            # Creates a GLfloat *
% glLightfv GL_LIGHT0 GL_AMBIENT $light          # Pass it to the function
...
% delete_fv4 $light                              # Destroy it (When done)
```

While not the most elegant approach, helper functions provide a simple mechanism for working with more complex datatypes. In most cases, they can be written without diving into SWIG's internals. Before typemap support was added to SWIG, helper functions were the only method for handling these kinds of problems. The pointer.i library file described earlier is an example of just this sort of approach. As a rule of thumb, I recommend that you try to use this approach before jumping into typemaps.

A Typemap Implementation

As we have seen, a typemap can often eliminate the need for helper functions. Without diving into the details, a more sophisticated typemap implementation of the previous example can permit you to pass an array or list of values directly into the C function like this :

```
% glLightfv GL_LIGHT0 GL_AMBIENT {0.0 0.0 0.0 1.0}
```

This is a more natural implementation that replaces the low-level pointer method. Now we will look into how one actually specifies a typemap.

What is a typemap?

A typemap is specified using the `%typemap` directive in your interface file. A simple typemap might look liks this :

```
%module example
%typemap(tcl,in) int {
        $target = atoi($source);
        printf(“Received an integer : %d\n”, $target);
}

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

In this case, we changed the processing of integers as input arguments to functions. When used in a Tcl script, we would get the following debugging information:

```
% set a [add 7 13]
Received an integer : 7
Received an integer : 13
```

In the typemap specification, the symbols `$source` and `$target` are holding places for C variable names that SWIG is going to use when generating wrapper code. In this example, `$source` would contain a Tcl string containing the input value and `$target` would be the C integer value that is going to be passed into the “add” function.

Creating a new typemap

A new typemap can be created as follows :

```
%typemap(lang,method) Datatype {
        ... Conversion code ...
}
```

`lang` specifies the target language, `method` defines a particular conversion method, and `Datatype` gives the corresponding C datatype. The code corresponding to the typemap is enclosed in braces after the declaration. There are about a dozen different kinds of typemaps that are used within SWIG, but we will get to that shortly.

A single conversion can be applied to multiple datatypes by giving a comma separated list of datatypes. For example :

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

Here, `$type` will be expanded into the real datatype during code generation. Datatypes may also carry names as in

```
%typemap(perl5,in) char **argv {
        ... Turn a perl array into a char ** ...
}
```

A “named”typemap will only apply to an object that matches both the C datatype and the name. Thus the `char **argv` typemap will only be applied to function arguments that exactly match “`char **argv`”. In some cases, the name may correspond to a function name (as is the case for return values).

Finally, there is a shortened form of the typemap directive :

```
%typemap(method) Datatype {
        ...
}
```

When the language name is ommitted, the typemap will be applied to the current target language. This form is only recommended for typemap methods that are language independent (there are a few). It is not recommended if you are building interfaces for multiple languages.

Deleting a typemap

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

```
%typemap(lang,method) Datatype;            // Deletes this typemap
```

Copying a typemap

A typemap can be copied using the following declaration :

```
%typemap(python,out) unsigned int = int;   // Copies a typemap
```

This specifies that the typemap for "`unsigned int`" should be the same as the "`int`" typemap. This is most commonly used when working with library files.

Typemap matching rules

When you specify a typemap, SWIG is going to try and match it with all future occurrences of the datatype you specify. The matching process is based upon the target language, typemap method, datatype, and optional name. Because of this, it is perfectly legal for multiple typemaps to exist for a single datatype at any given time. For example :

```
%typemap(tcl,in) int * {
       ... Convert an int * ...
}
%typemap(tcl,in) int [4] {
       ... Convert an int[4] ...
}
%typemap(tcl,in) int out[4] {
       ... Convert an out[4] ...
}
%typemap(tcl,in) int *status {
       ... Convert an int *status ...
}
```

These typemaps all involve the "`int *`" datatype in one way or another, but are all considered to be distinct. There is an extra twist to typemaps regarding the similarity between C pointers and arrays. A typemap applied to a pointer will also work for any array of the same type. On the other hand, a typemap applied to an array will only work for arrays, not pointers. Assuming that you're not completely confused at this point, the following rules are applied in order to match pointers and arrays :

- Named arrays
- Unnamed arrays
- Named datatypes
- Unnamed datatypes

The following interface file shows how these rules are applied.

```
void foo1(int *);           // Apply int * typemap
void foo2(int a[4]);        // Apply int[4] typemap
void foo3(int out[4]);      // Apply int out[4] typemap
void foo4(int *status);     // Apply int *status typemap
void foo5(int a[20]);       // Apply int * typemap (because int [20] is an int *)
```

Because SWIG uses a name-based approach, it is possible to attach special properties to named parameters. For example, we can make an argument of "`int *OUTPUT`" always be treated as an output value of a function or make a "`char **argv`" always accept a list of string values.

Common typemap methods

The following methods are supported by most SWIG language modules. Individual language may provide any number of other methods not listed here.

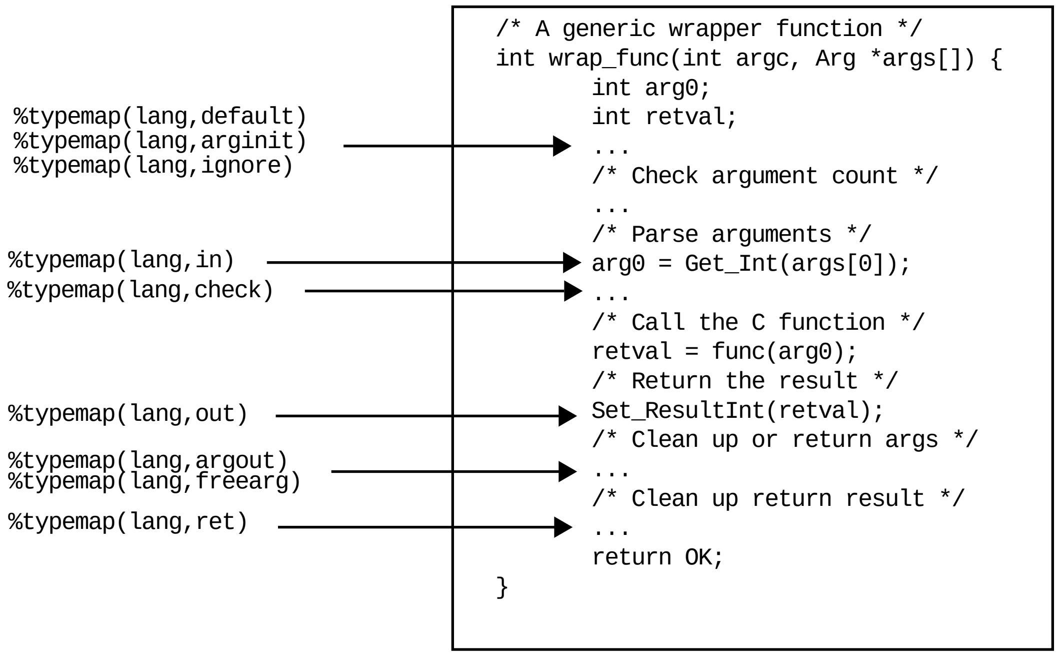
Common Typemap Methods

`%typemap(lang,in)`	Convert function arguments from the scripting language to a C representation.
`%typemap(lang,out)`	Converts the return value from a function to a scripting language representation.
`%typemap(lang,ret)`	Cleans up the return value of a function. For example, this could be used to free up memory that might have been allocated by the underlying C function. This code is executed before a function returns control back to the scripting language.
`%typemap(lang,freearg)`	Cleans up arguments. This method can be used to clean up function arguments that might have required memory allocation or other special processing.
`%typemap(lang,argout)`	Outputs argument values. This typemap can be used to make a function return a value from one of its arguments.
`%typemap(lang,check)`	Checks validity of function inputs. Can be used to apply constraints, raise exceptions, or simply for debugging.
`%typemap(lang,varin)`	Used by some languages to set the value of a C global variable. Converts a datatype from the scripting language to C. This method is only used if the "in" method won't work for some reason.
`%typemap(lang,varout)`	Variable. Convert the value of a C global variable to a scripting language representation.
`%typemap(lang,const)`	Specifies the code used to create a constant in the module initialization function. Not supported by all languages.
`%typemap(lang,memberin)`	Set structure member. Specifies special processing of structure and class members when setting a value.
`%typemap(lang,memberout)`	Get structure member. Special processing applied when retrieving a structure member.

Common Typemap Methods

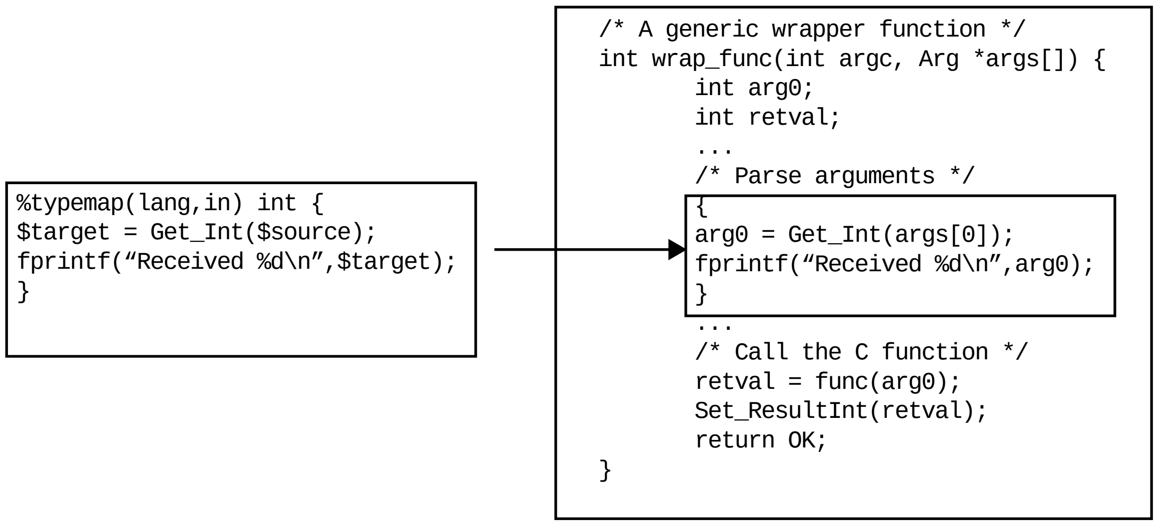
`%typemap(lang,arginit)`	Initializes a parameter to an initial value (for example, setting a pointer to a NULL value). Sometimes useful in determining whether a parameter was received correctly.
`%typemap(lang,default)`	Can be used to set a default argument value. Overrides any other default arguments that might have been specified.
`%typemap(lang,ignore)`	Set an argument to a default value and ignore it for the purposes of generating wrapper code. (An ignored argument becomes a hidden argument in the scripting interface).

Understanding how some of these methods are applied takes a little practice and better understanding of what SWIG does when it creates a wrapper function. The next few diagrams show the anatomy of a wrapper function and how the typemaps get applied. More detailed examples of typemaps can be found on the chapters for each target language.



```
/* A generic wrapper function */
int wrap_func(int argc, Arg *args[]) {
       int arg0;
       int retval;
       ...
       /* Check argument count */
       ...
       /* Parse arguments */
       arg0 = Get_Int(args[0]);
       ...
       /* Call the C function */
       retval = func(arg0);
       /* Return the result */
       Set_ResultInt(retval);
       /* Clean up or return args */
       ...
       /* Clean up return result */
       ...
       return OK;
}
```

Wrapper function typemaps and where they are applied



```
%typemap(lang,in) int {
$target = Get_Int($source);
fprintf("Received %d\n",$target);
}
```

```
/* A generic wrapper function */
int wrap_func(int argc, Arg *args[]) {
       int arg0;
       int retval;
       ...
       /* Parse arguments */
       {
       arg0 = Get_Int(args[0]);
       fprintf("Received %d\n",arg0);
       }
       ...
       /* Call the C function */
       retval = func(arg0);
       Set_ResultInt(retval);
       return OK;
}
```

How a typemap gets applied to a wrapper function

Writing typemap code

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

Scope

Typemap code is 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. However, local variables will only exist in the tiny portion of code you supply. In other words, any local variables that you create in a typemap will disappear when the typemap has completed its execution.

Creating local variables

Sometimes it is necessary 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(tcl,in) int *INPUT(int temp) {
       temp = atoi($source);
       $target = &temp;
}
```

What happens here is that `temp` becomes a local variable in the scope of the entire wrapper function. When we set it to a value, that values persists for the duration of the wrapper function and gets cleaned up automatically on exit. This is particularly useful when working with pointers and temporary values.

It is perfectly safe to use multiple typemaps involving local variables in the same function. For example, we could declare a function as :

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

When this occurs, SWIG will create three different local variables named ‘temp’. Of course, they don’t all end up having the same name---SWIG automatically performs a variable renaming operation if it detects a name-clash like this.

Some typemaps do not recognize local variables (or they may simply not apply). At this time, only the “in”, “argout”, “default”, and “ignore” typemaps use local variables.

Special variables

The following special variables may be used within a typemap conversion code :

Special variables

`$source`	C Variable containing source value
`$target`	C Variable where result is to be placed
`$type`	A string containing the datatype (will be a pointer if working with arrays).
`$basetype`	A string containing the base datatype. In the case of pointers, this is the root datatype (without any pointers).
`$mangle`	Mangled string representing the datatype (used for creating pointer values)
`$value`	Value of contants (const typemap only)
`$argnum`	Argument number (functions only)
`$arg`	Original argument. Often used by typemaps returning v alues through function parameters.
`$name`	Name of C declaration (usually the name of a function).
`$<op>`	Insert the code for any previously defined typemap. <op> must match the name of the the typemap being defined (ie. “in”, “out”, “argout”, etc...). This is used for typemap chaining and is not recommended unless you absolutely know what you’re doing.

When found in the conversion code, these variables will be replaced with the correct values. Not all values are used in all typemaps. Please refer to the SWIG reference manual for the precise usage.

Typemaps for handling arrays

One of the most common uses of typemaps is providing some support for arrays. Due to the subtle differences between pointers and arrays in C, array support is somewhat limited unless you provide additional support. For example, consider the following structure appears in an interface file :

```
struct Person {
       char name[32];
       char address[64];
```

```
        int id;
};
```

When SWIG is run, you will get the following warnings :

```
swig -python  example.i
Generating wrappers for Python
example.i : Line 2. Warning. Array member will be read-only.
example.i : Line 3. Warning. Array member will be read-only.
```

These warning messages indicate that SWIG does not know how you want to set the name and address fields. As a result, you will only be able to query their value.

To fix this, we could supply two typemaps in the file such as the following :

```
%typemap(memberin) char [32] {
        strncpy($target,$source,32);
}
%typemap(memberin) char [64] {
        strncpy($target,$source,64);
}
```

The "memberin" typemap is used to set members of structures and classes. When you run the new version through SWIG, the warnings will go away and you can now set each member. It is important to note that `char[32]` and `char[64]` are different datatypes as far as SWIG typemaps are concerned. However, both typemaps can be combined as follows :

```
// A better typemap for char arrays
%typemap(memberin) char [ANY] {
        strncpy($target,$source,$dim0);
}
```

The `ANY` keyword can be used in a typemap to match any array dimension. When used, the special variable `$dim0` will contain the real dimension of the array and can be used as shown above.

Multidimensional arrays can also be handled by typemaps. For example :

```
// A typemap for handling any int [][] array
%typemap(memberin) int [ANY][ANY] {
        int i,j;
        for (i = 0; i < $dim0; i++)
                for (j = 0; j < $dim1; j++) {
                        $target[i][j] = *($source+$dim1*i+j);
                }
}
```

When multi-dimensional arrays are used, the symbols `$dim0`, `$dim1`, `$dim2`, etc... get replaced by the actual array dimensions being used.

The ANY keyword can be combined with any specific dimension. For example :

```
%typemap(python,in) int [ANY][4] {
        ...
}
```

A typemap using a specific dimension always has precedence over a more general version. For example, `[ANY][4]` will be used before`[ANY][ANY]`.

Typemaps and the SWIG Library

Writing typemaps is a tricky business. For this reason, many common typemaps can be placed into a SWIG library file and reused in other modules without having to worry about nasty underlying details. To do this, you first write a file containing typemaps such as this :

```
// file : stdmap.i
// A file containing a variety of useful typemaps

%typemap(tcl,in) int INTEGER {
       ...
}
%typemap(tcl,in) double DOUBLE {
       ...
}
%typemap(tcl,out) int INT {
       ...
}
%typemap(tcl,out) double DOUBLE {
       ...
}
%typemap(tcl,argout) double DOUBLE {
       ...
}
// and so on...
```

This file may contain dozens or even hundreds of possible mappings. Now, to use this file with other modules, simply include it in other files and use the `%apply` directive :

```
// interface.i
// My interface file

%include stdmap.i                          // Load the typemap library

// Now grab the typemaps we want to use
%apply double DOUBLE {double};

// Rest of your declarations
```

In this case, `stdmap.i` contains a variety of standard mappings. The %apply directive lets us apply specific versions of these to new datatypes without knowing the underlying implementation details.

To clear a typemap that has been applied, you can use the %clear directive. For example :

```
%clear double x;     // Clears any typemaps being applied to double x
```

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(perl5,check) double *posdouble {
        if ($target < 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(python,check) Vector * {
        if ($target == 0) {
                PyErr_SetString(PyExc_TypeError,”NULL Pointer not allowed”);
                return NULL;
        }
}
```

will prevent any function involving a `Vector *` from accepting a NULL pointer. As a result, SWIG can often prevent a potential segmentation faults or other run-time problems by raising an exception rather than blindly passing values to the underlying C/C++ program.

Typemap examples

Typemaps are inherently language dependent so more examples appear in later chapters. The SWIG `Examples` directory also includes a variety of examples. Sophisticated users may gain more by examining the `typemaps.i` and `constraints.i` SWIG library files.

How to break everything with a typemap

It should be emphasized that typemaps provide a direct mechanism for modifying SWIG’s output. As a result, it can be very easy to break almost everything if you don’t know what you’re doing. For this reason, it should be stressed that typemaps are not required in order to use SWIG with most kinds of applications. Power users, however, will probably find typemaps to be a useful tool for creating extremely powerful scripting language extensions.

Typemaps and the future

The current typemap mechanism, while new, will probably form the basis of SWIG 2.0. Rather than having code buried away inside a C++ module, it may soon be possible to redefine almost all of SWIG’s code generation on the fly. Future language modules will rely upon typemaps almost exclusively.

Exception Handling 7

In some cases, it is desirable to catch errors that occur in C functions and propagate them up to the scripting language interface (ie. raise an exception). By default, SWIG does nothing, but you can create a user-definable exception handler using the `%except` directive.

The %except directive

The `%except` directive allows you to define an exception handler. It works something like this :

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

As an argument, you need to specify the target language. The exception handling C/C++ code is then enclosed in braces. The symbol `$function` is replaced with the real C/C++ function call that SWIG would be ordinarily make in the wrapper code. The C code you specify inside the `%except` directive can be anything you like including custom C code and C++ exceptions.

To delete an exception handler, simply use the `%except` directive with no code. For example :

```
%except(python);              // Deletes any previously defined handler
```

Exceptions can be redefined as necessary. The scope of an exception handler is from the point of definition to the end of the file, the definition of a new exception handler, or until the handler is deleted.

Handling exceptions in C code

C has no formal mechanism for handling exceptions so there are many possibilities. The first approach is to simply provide some functions for setting and checking an error code. For example :

```
/* File : except.c */
```

```
static char error_message[256];
static int error_status = 0;

void throw_exception(char *msg) {
       strncpy(error_message,msg,256);
       error_status = 1;
}

void clear_exception() {
       error_status = 0;
}
char *check_exception() {
       if (error_status) return error_message;
       else return NULL;
}
```

To work, functions will need to explicitly 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) :

```
%except(perl5) {
       char *err;
       clear_exception();
       $function
       if ((err = check_exception())) {
              croak(err);
       }
}
```

Now, when an error occurs, it will be translated into a Perl error. The downside to this approach is that it isn't particularly clean and it assumes that your C code is a willing participant in generating error messages. (This isn't going to magically add exceptions to a code that doesn't have them).

Exception handling with longjmp()

Exception handling can also be added to C code using the `<setjmp.h>` library. This usually isn't documented very well (at least not in any of my C books). In any case, here's one implementation that uses 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"
%}

%except(perl5) {
       try {
              $function
       } catch(RangeError) {
              croak("Range Error");
       } catch(DivisionByZero) {
              croak("Division by zero");
       } catch(OutOfMemory) {
              croak("Out of memory");
       } finally {
              croak("Unknown exception");
       }
}
```

At this point, you're saying this sure looks alot like C++ and you'd be right (C++ exceptions are often implemented in a similar manner). As always, the usual disclaimers apply--your mileage may vary.

Handling C++ exceptions

Handling C++ exceptions is almost completely trivial (well, all except for the actual C++ part). A typical SWIG exception handler will look like this :

```
%except(perl5) {
```

```
                try {
                        $function
                } 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 {};
```

Newer versions of the SWIG parser should ignore exceptions specified in function declarations. For example :

```
        double inv(double) throw(DivisionByZero);
```

Defining different exception handlers

By default, the `%except` directive creates an exception handler that is used for all wrapper functions that follow it. Creating one universal exception handler for all functions may be unwieldy and promote excessive code bloat since the handler will be inlined into each wrapper function created. For this reason, the exception handler can be redefined at any point in an interface file. Thus, a more efficient use of exception handling may work like this :

```
        %except(python) {
                ... your exception handler ...
        }
        /* Define critical operations that can throw exceptions here */

        %except(python);          // Clear the exception handler

        /* Define non-critical operations that don't throw exceptions */
```

Applying exception handlers to specific datatypes.

An alternative approach to using the `%except` 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(python,except) void * {
                $function
                if (!$source) {
                        PyExc_SetString(PyExc_MemoryError,"Out of memory in $name");
                        return NULL;
                }
```

```
}

void *malloc(int size);
```

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

```
Python 1.4 (Jan 16 1997)  [GCC 2.7.2]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from example import *
>>> a = malloc(2048)
>>> b = malloc(1500000000)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
MemoryError: Out of memory in malloc
>>>
```

Since typemaps can be named, you can define an exception handler for a specific function as follows :

```
%typemap(python,except) void *malloc {
       ...
}
```

This will only be applied to the `malloc()` function returning `void *`. While you probably wouldn't want to write a different exception handler for every function, it is possible to have a high degree of control if you need it. When typemaps are used, they override any exception handler defined with `%except`.

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 scripting language exceptions in a portable manner. For example :

```
// Language independent exception handler
%include exception.i

%except {
       try {
              $function
       } 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. The exception library provides a language-independent exception handling mechanism, so many of SWIG's library files now rely upon the library as well.

Debugging and other interesting uses for %except

Since the `%except` directive allows you to encapsulate the actual C function call, it can also be used for debugging and tracing operations. For example :

```
%except(tcl) {
       printf("Entering function : $name\n");
       $function
       printf("Leaving function : $name\n");
}
```

allows you to follow the function calls in order to see where an application might be crashing.

Exception handlers can also be chained. For example :

```
%except(tcl) {
       printf("Entering function : $name\n");
       $except
       printf("Leaving function : $name\n");
}
```

Any previously defined exception handler will be inserted in place of the "`$except`" symbol. As a result, you can attach debugging code to existing handlers if necessary. However, it should be noted that this must be done before any C/C++ declarations are made (as exception handlers are applied immediately to all functions that follow them).

More Examples

By now, you know most of the exception basics. See the SWIG Examples directory for more examples and ideas. Further chapters show how to generate exceptions in specific scripting languages.

SWIG and Perl5 8

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
```

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 you 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 `permain.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
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 /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
```

Usually, a module consists of a collection of code that is contained within a single file. A package, on the other hand, is the Perl equivalent of a namespace. A package is alot like a module, except that it is independent of files. Any number of files may be part of the same package--or a package may be broken up into a collection of modules if you prefer to think about it in this way.

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);
```

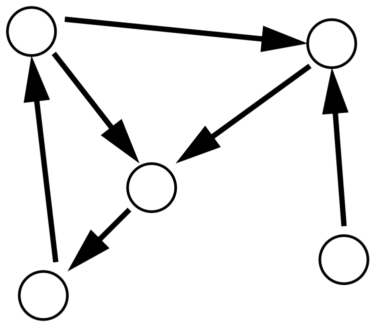
To use the class, simply use these functions. As it turns out, SWIG has a mechanism for creating shadow classes that hides these functions and uses an object oriented interface instead--keep reading.

A simple Perl example

In the next few sections, use of the Perl5 module will be illustrated through a series of increasingly complex examples. In the example, we will write some simple graph algorithms to illustrate how C and Perl can interact with each other.

Graphs

A directed graph is simply a collection of nodes (or vertices) that are connected to each other by a collection of edges :



A graph

To represent simple graphs, we can use the following C data structures :

```
/* File : graph.h */
/* Simple data structures for directed graph of Nodes and Edges */

struct Edge;
typedef struct Node {
  int           v;            /* Vertex number              */
  struct Edge  *adj;          /* Adjacency List             */
} Node;

typedef struct Edge {
  Node          *node;        /* Connecting Node            */
  double            w;        /* Weight (optional)          */
  struct Edge   *next;        /* Next edge                  */
} Edge;
```

Each node contains a unique number v for identifying it, and a linked list of other nodes that are nearby. This linked list is managed by the `Edge` structure. Associated with each edge is an optional weighting factor. This could be something like miles between cities, bandwidth, or some other numerical property about a particular link. Okay, enough talk about data structures for now.

To construct nodes and add edges, we can use the following C code :

```
/* File : graph.c           */
/* Directed graph functions */

#include "graph.h"
static int node_count = 0;          /* Number of nodes created */
```

```
/* Create a new Node */
Node *new_Node() {
  Node *node;
  node = (Node *) malloc(sizeof(Node));
  node->v = node_count++;       /* Bump up the count */
  node->adj = (Edge *) 0;
  return node;
}

/* Add an "edge" to another node. */

Edge *Node_addedge(Node *mynode, Node *othernode, double w) {
  Edge *edge;
  edge = (Edge *) malloc(sizeof(Edge));
  edge->node = othernode;
  edge->w = w;
  edge->next = mynode->adj;    /* add to my adjacency list */
  mynode->adj = edge;
  return edge;
}
```

A simple SWIG interface file

So far, the code doesn't do much, but it's enough to wrap into Perl5 module. Since the C code is fairly simple right now, we can do this by creating the following interface file :

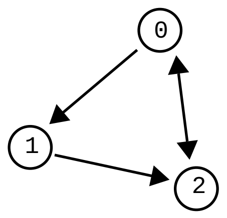
```
%module graph
%{
#include "graph.h"
%}
%include graph.h             // Get structure definition
%include graph.c             // Get support functions
```

We'll call our module "graph" and simply read in both the files `graph.h` and `graph.c` to build an interface.

Sample Perl Script

Once compiled, we can now start using our new module. All of the C functions found in `graph.h` and `graph.c` can be called normally . Even though we are using pointers, this is not a problem. Here is a very simple script :

```
# Perl code to use our graph code

use graph;
package graph;

# Create some nodes
$n0 = new_Node();
$n1 = new_Node();
$n2 = new_Node();

# Make some edges
Node_addedge($n0,$n1,0);        #  0 -> 1
Node_addedge($n0,$n2,0);        #  0 -> 2
```



```
Node_addedge($n1,$n2,0);        #  1 -> 2
Node_addedge($n2,$n0,0);        #  2 -> 0

# A procedure to print out a node and its adjacency list
sub print_node {
       my $node = shift;
       print "Node : ", Node_v_get($node), ", Adj : ";
       my $adj = Node_adj_get($node);
       while (defined($adj)) {                   # This checks for a NULL pointer
              my $anode = Edge_node_get($adj);
              my $v = Node_v_get($anode);
              print "$v ";
              $adj = Edge_next_get($adj);        # Move to next node
       }
       print "\n";
}

# Print out node information
print_node($n0);
print_node($n1);
print_node($n2);
```

When executed, this script produces the following output :

```
Node : 0, Adj : 2 1
Node : 1, Adj : 2
Node : 2, Adj : 0
```

While our two C functions are used in the script, SWIG also created a collection of accessor functions for managing the two C data structures. The functions `Node_v_get()`, `Node_adj_get()`, `Edge_node_get()`, and `Edge_next_get()` are used to access the corresponding members of our `Node` and `Edge` structures. As arguments, these functions simply take a pointer to the corresponding structure.

There are a few other things to notice about the code.

- Pointers to complex objects are manipulated like ordinary scalar values, but in reality they are blessed references. For all practical purposes, you should think of them as funny opaque objects (in other words, don't worry about it).
- To check for a NULL pointer, use the expression `defined($ptr)`. This will return true if a pointer is non-NULL, false if it isn't. In our example, we use this to walk down a linked list of pointers until we reach a NULL value.

Even though the original C code was rather useless by itself, we have used it to build a simple graph in Perl along with a debugging function for printing out node information. In fact, without making any modifications to the C code, we can use this to build up something more complex such as a database of cities and mileages.

```
Denver, Cheyenne,           100
Denver, Santa Fe,           362
Denver, Kansas City,        600
Santa Fe, Albuquerque,      55
Santa Fe, Durango,          214
...
Durango, Moab,              160
Moab, Salt Lake City,       237
Moab, Denver,               310
Cheyenne, Salt Lake City,   436
```

Some cities and mileages

Here's a slightly more complicated Perl script to read in the above mileage table and turn it into a graph:

```
# Read a file with cities into a graph

use graph;
package graph;

%Cities = ();              # Hash table mapping cities to nodes
%Nodes = ();               # Mapping of Node indicies to cities
sub read_cities {
    my $filename = shift;
    open(CITIES,$filename);
    while (<CITIES>) {
              chop;
              my @a = split(/, +/);
              my $node1;
              my $node2;
              # Check to see if a given city is already a node
              if (!exists $Cities{$a[0]}) {
                     $node1 = new_Node();
                     $Cities{$a[0]} = $node1;
                     my $node_num = Node_v_get($node1);
                     $Nodes{$node_num} = $a[0];
              } else {
                     $node1 = $Cities{$a[0]};
              }
              if (!exists $Cities{$a[1]}) {
                     $node2 = new_Node();
                     $Cities{$a[1]} = $node2;
                     my $node_num = Node_v_get($node2);
                     $Nodes{$node_num} = $a[1];
              } else {
                     $node2 = $Cities{$a[1]};
              }
              # Add edges
              Node_addedge($node1,$node2,$a[2]);
              Node_addedge($node2,$node1,$a[2]);
       }
}
```

```
sub print_near {
    my $city = shift;
    if (exists $Cities{$city}) {
            my $node = $Cities{$city};
            print "Cities near $city : ";
            my $adj = Node_adj_get($node);
            while (defined($adj)) {
                    my $anode = Edge_node_get($adj);
                    my $v = Node_v_get($anode);
                    print $Nodes{$v},", ";
                    $adj = Edge_next_get($adj);
            }
    }
    print "\n";
}
read_cities("cities");
print_near("Denver");
print_near("Las Vegas");
```

This produces the following output :

```
Cities near Denver : Moab, Kansas City, Santa Fe, Cheyenne, Albuquerque,
Cities near Las Vegas : Flagstaff, Los Angeles, Moab, Salt Lake City,
```

In this example, we are using the same functions as before, but we are now introducing two Perl hash tables. The `%Cities` hash is used to associate city names with a corresponding node in the graph. The `%Nodes` hash does exactly the opposite---mapping node numbers back to the names of cities. Both of these will come in quite handy for mapping things between the Perl world and the C world later.

Before proceeding, let's stop and summarize what we have done. Given a couple of very simple C data structures for a graph, we have written a program that can read in a mileage table, construct a weighted graph of the data and print out a list of the cities that are nearby other cities. Yet, the C code knows nothing about the Perl interface or this whole mileage program we've built up around it. While we could have written the entire program in C, we would have had to write a main program, some code to read the input file, and a hash table structure to keep track of the mapping between nodes and cities. Perl, on the other hand, is very good at managing these tasks.

Accessing arrays and other strange objects

Now let's add some new functionality to our graph program from the previous example. In this case, we'll add a depth-first search algorithm to see if two nodes are connected to each other (possibly through many other nodes in-between).

We'll first add the following constants to the file `graph.h`

```
/* File : graph.h */
...
#define MAX_NODES   1000
#define UNSEEN      -1
```

Now, a modified version of `graph.c` :

```
/* File : graph.c */
/* Directed graph functions */

#include <stdio.h>
#include "graph.h"

int node_count = 0;          /* Number of nodes created     */
int seen[MAX_NODES];         /* Used by the search function */

...

/* Function to search for node with given vertex number */
static int visit(Node *n,int v) {
  Edge *e;

  if (seen[n->v] != UNSEEN) return UNSEEN;  /* Cycle detected */
  if (n->v == v) return 1;                  /* Match found    */
  e = n->adj;
  while (e) {
    seen[n->v] = e->node->v;
    if (visit(e->node,v) > 0) return 1;
    e = e->next;
  }
  return 0;
}


/* Depth-first search for a given node */
int Node_search(Node *start, int v) {
  int i;

  for (i = 0; i < node_count; i++)
    seen[i] = UNSEEN;
  return visit(start,v);
}
```

The idea here is simple, the function `Node_search()` takes a starting node and starts looking for a node with given vertex. Upon startup, the search function clears an array of values indicating whether a node has been seen or not. While this array is primarily used to detect cycles, it can also be used to store the path between two nodes as we proceed through the algorithm. Upon exit, we can then use the array to figure out how we got between the starting and ending node. Of course, this leads us to the question of how we access this array in Perl.

As a general rule, handling arrays is somewhat problematic since the mapping between arrays and pointers may not be what you expect (even in C) and there is not necessarily a natural mapping between arrays in C and arrays in Perl (for example, if we've got a C array with 1 million elements in it, we almost certainly wouldn't want to convert it to a Perl array!).

To access our array, we will write a C helper function that allows us to access invidual elements. However, rather than adding this function to the C code, we can insert it directly into our SWIG interface file. We will also strip the function names out of the `.c` file and put their prototypes in the header file :

```
%module graph
```

```
%{
#include "graph.h"
%}
%include graph.h

%inline %{
/* Get seen value for a particular node  */
int get_seen(int index) {
  extern int node_count;
  extern int seen[];
  if ((index < 0) || (index >= node_count)) return -1;
  else return seen[index];
}
%}
```

This interface file illustrates one of the key points about SWIG--even though SWIG uses C syntax, wrapping arbitrary C code doesn't always result in a good interface. Almost any significant package will require the use of a few "helper" functions to get at certain data structures or to change the way in which a function is called.

With our new C search function, we can now write a Perl function to find a route between two cities. This function simply takes the names of two cities, uses the `Cities` hash to look up their nodes and calls the C `Node_search()` function. Afterwards, we walk through the `seen[]` array using our helper function and print the route.

```
sub find_route {
    my $start = shift;
    my $dest = shift;
    # Lookup nodes from names
    if ((!exists $Cities{$start}) || (!exists $Cities{$dest})) {
       return;
    }
    my $node1 = $Cities{$start};
    my $node2 = $Cities{$dest};
    print "$start --> $dest :\n";

    # Depth first search for a route between cities
    my $found = Node_search($node1,Node_v_get($node2));
    if ($found) {
       $v = Node_v_get($node1);
       while ($v != $UNSEEN) {
           print "    $Nodes{$v}\n";
           $v = get_seen($v);
       }
    } else {
       print "    You can't get there from here\n";
    }
}

read_cities("cities");
find_route("Salt Lake City","Denver");
```

Of course, depth first search isn't very good at finding an optimal route---obviously this output must be the very scenic route!

```
Salt Lake City --> Denver :
```

```
Salt Lake City
Twin Falls
Boise
Portland
Eugene
San Francisco
Los Angeles
Las Vegas
Flagstaff
Albuquerque
Santa Fe
Durango
Moab
Denver
```

Implementing methods in Perl

Obviously, our depth search algorithm isn't so useful in this specific application. Perhaps we would like to try a breadth-first search algorithm instead. We could choose to write it in C, but the breadth first search algorithm depends on the use of a queue to hold the list of nodes to visit. Thus, we'd have to write a queue data structure first. However, a Perl array smells alot like a queue if we manipulate it in the right way. So we can use Perl to come up with a quick and dirty breadth first search without dropping down into C :

```
%visit = ();
sub breadth_search {
    my $node1 = shift;
    my $node2 = shift;
    my @queue;
    %visit = ();
    # Put starting node into queue
    push @queue, $node1;
    $visit{Node_v_get($node1)} = Node_v_get($node1);
    while (@queue) {            # Loop until queue is empty
       my $n = shift @queue;     # Pop off an node
       my $nv = Node_v_get($n);
       return 1 if ($$n == $$node2);   # Exit if we found the destination
       # Put children onto the queue
       my $e = Node_adj_get($n);
       while (defined($e)) {
           my $m = Edge_node_get($e);
           my $v = Node_v_get($m);
           if (!exists $visit{$v}) {
              push @queue, $m;
              $visit{$v} = $nv;
           }
           $e = Edge_next_get($e);
       }
    }
    return 0;
}

sub find_route {
    my $start = shift;
    my $dest = shift;
    # Lookup nodes from names
```

```
        return if ((!exists $Cities{$start}) || (!exists $Cities{$dest}));
        print "$start --> $dest :\n";
        my $node1 = $Cities{$start};
        my $node2 = $Cities{$dest};
        my $found = breadth_search($node1,$node2);
        my @path;
        if ($found) {
           my $v = Node_v_get($node2);
           delete $visit{Node_v_get($node1)};
           while (exists($visit{$v})) {
               unshift @path,$Nodes{$v};
               $v = $visit{$v};
           }
           unshift @path,$start;
           foreach $c (@path) { print "    $c\n";}
        } else {
           print "    You can't get there from here\n";
        }
    }
```

Our Perl implementation creates a queue using an array and manipulating it with shift and push operations. The global hash `%visit` is used to detect cycles and to determine how we got to each node. When we find a route, we can march backwards through the route to determine the entire path. When we run our new code, we get the following :

```
find_route("Salt Lake City", "Denver");
Salt Lake City --> Denver :
    Salt Lake City
    Cheyenne
    Denver
```

Clearly this is a more efficient route--although admittedly not very scenic. If we wanted to get even more serious, we could add a priority search based on mileages. Later on we might implement these features in C for better performance. Either way, it is reasonably easy to manipulate complex structures in Perl and to mix them with C code (well, with a little practice perhaps).

Shadow classes

By now, you've probably noticed that examples have been using alot of accessor functions to get at the members of our `Node` and `Edge` structures. This tends to make the Perl code look rather cluttered (well, more than normal Perl code in any case) and it isn't very object oriented.

With a little magic, SWIG can turn C structs and C++ classes into fully functional Perl classes that work in a more-or-less normal fashion. This transformation is done by writing an additional Perl layer that builds Perl classes on top of the low-level SWIG interface. These Perl classes are said to "shadow" an underlying C/C++ class.

To have SWIG create shadow classes, use the `-shadow` option :

```
% swig -perl5 -shadow graph.i
```

This will produce the same files as before except that the `.pm` file will now contain significantly more supporting Perl code. While there are some rather subtle aspects of this transformation, for now we'll omit the details and illustrate the changes in an example first (the use of shadow classes has been underlined)

```
# Read a file with cities into a graph
# Uses shadow classes

use graph;
package graph;

%Cities = ();              # Hash table mapping cities to nodes
%Nodes = ();               # Mapping of Node indicies to cities

sub read_cities {
    my $filename = shift;
    open(CITIES,$filename);
    while (<CITIES>) {
       chop;
       my @a = split(/, +/);
       my $node1;
       my $node2;
       # Check to see if a given city is already a node
       if (!exists $Cities{$a[0]}) {
           $node1 = new_Node();
           $Cities{$a[0]} = $node1;
           $Nodes{$node1->{v}} = $a[0];      # Note access of 'v'
       } else {
           $node1 = $Cities{$a[0]};
       }
       if (!exists $Cities{$a[1]}) {
           $node2 = new_Node;
           $Cities{$a[1]} = $node2;
           $Nodes{$node2->{v}} = $a[1];
       } else {
           $node2 = $Cities{$a[1]};
       }
       # Add edges
       Node_addedge($node1,$node2,$a[2]);
       Node_addedge($node2,$node1,$a[2]);
    }
}

%visit;
sub breadth_search {
    my $node1 = shift;
    my $node2 = shift;
    my @queue;
    %visit = ();
    my $dest = $node2->{v};
    # Put starting node into queue
    push @queue, $node1;
    $visit{$node1->{v}} = $node1->{v};
    while (@queue) {
       my $n = shift @queue;
       return 1 if ($n->{v} == $node2->{v});
       # Put children onto the queue
       my $e = $n->{adj};
       while (defined($e)) {
           if (!exists $visit{$e->{node}->{v}}) {
                push @queue, $e->{node};
                $visit{$e->{node}->{v}} = $n->{v};
            }
            $e = $e->{next};
        }
    }
    return 0;
}

sub find_route {
    my $start = shift;
    my $dest = shift;
    # Lookup nodes from names
    return if ((!exists $Cities{$start}) || (!exists $Cities{$dest}));
    print "$start --> $dest :\n";
    my $node1 = $Cities{$start};
    my $node2 = $Cities{$dest};
    my $found = breadth_search($node1,$node2);
    my @path;
    if ($found) {
       my $v = $node2->{v};
       delete $visit{$node1->{v}};
       while (exists($visit{$v})) {
           unshift @path,$Nodes{$v};
           $v = $visit{$v};
       }
       unshift @path,$start;
       foreach $c (@path) {
           print "    $c\n";
       }
    } else {
       print "    You can't get there from here\n";
    }
}
read_cities("cities");

find_route("Salt Lake City","Denver");
```

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 accessor functions. like before. However, this example is only scratching the surface of what can be done with shadow classes...keep reading.

Getting serious

Now that we've got a very simple example working, it's time to get really serious. Suppose that in addition to working with the mileage data of various cities, we want to make a graphical representation from geographical data (lattitude/longitude). To do this, we'll use SWIG to glue together a bunch of stuff. First, for the purposes of illustration, let's create a new C data structure for holding a geographical location with the assumption that we might want to use it in some C functions later :

```
/* File : location.h */
/* Data structure for holding longitude and lattitude information */

typedef struct Location {
  char    *name;
  double  lat_degrees;
  double  lat_minutes;
  double  lat_seconds;
  char    lat_direction;
  double  long_degrees;
  double  long_minutes;
  double  long_seconds;
  char    long_direction;
} Location;

extern Location *new_Location(char *name);
```

We also probably want a C function for creating one of these objects:

```
/* File : location.c */
#include <string.h>

/* Make a new location */
Location *new_Location(char *name) {
  Location *l;
  l = (Location *) malloc(sizeof(Location));
  l->name = (char *) malloc(strlen(name)+1);
  strcpy(l->name,name);
  return l;
}
```

Now let's make an interface file for this module :

```
// File : location.i
%module location
%{
#include “location.h”
%}

%include location.h
```

We could use this interface file to make an entirely new Perl5 module or we can combine it with the graph module. In this latter case, we simply need to put “`%include location.i`” in the file `graph.i`.

Now, finally, we could write a Perl function to read data in the following format :

```
Santa Fe,       35 41 13 N 105 56 14 W
Denver,         39 44 21 N 104 59 03 W
Albuquerque,    35 05 00 N 106 39 00 W
Cheyenne,       41 08 00 N 104 49 00 W
Kansas City,    39 05 51 N 94 37 38 W
Durango,        37 16 31 N 107 52 46 W
Moab,           38 34 24 N 109 32 57 W
Salt Lake City, 40 45 39 N 111 53 25 W
Reno,           39 31 47 N 119 48 46 W
San Francisco,  37 46 30 N 122 25 06 W
Las Vegas,      36 10 30 N 115 08 11 W
Flagstaff,      35 11 53 N 111 39 02 W
Los Angeles,    34 03 08 N 118 14 34 W
Eugene,         44 03 08 N 123 05 08 W
Portland,       45 31 24 N 122 40 30 W
Seattle,        47 36 23 N 122 19 51 W
Boise,          43 36 49 N 116 12 09 W
Twin Falls,     42 33 47 N 114 27 36 W
```

Geographic data

```
sub read_locations {
    my $filename = shift;
    open(LOCATIONS,$filename);
    while (<LOCATIONS>) {
       chop;
       my @a = split(/, +/);
       my $loc;
       # Check to see if this is a city I know about
       if (exists $Cities{$a[0]}) {
           # Create a new location
           $loc = new_Location($a[0]);
           my @coords = split(' ',$a[1]);
           # A nice way to assign attributes to an object
           %$loc = (lat_degrees => $coords[0],
                    lat_minutes => $coords[1],
                    lat_seconds => $coords[2],
                    lat_direction => $coords[3],
                    long_degrees => $coords[4],
                    long_minutes => $coords[5],
                    long_seconds => $coords[6],
                    long_direction => $coords[7]);
           my $v = $Cities{$a[0]}->{v};
           $Locations{$v} = $loc;
       }
    }
    close LOCATIONS;
}
```

Again, we are using shadow classes which are allowing us to assign all of the attributes of a C structure in the same way as one would create a Perl hash table. We have also created the `%Locations` hash to associate node numbers with a given location object.

Of course, having locations isn't too useful without a way to look at them so we'll grab the public domain `gd` library by Thomas Boutell. First, we'll write a simple C function to draw two locations and draw a line between them (some code has been omitted for clarity) :.

```
/* File : plot.c */
```

```
#include <gd.h>
#include <gdfonts.h>
#include "location.h"
double xmin,ymin,xmax,ymax;   /* Plotting range */

/* Make a plot of two locations with a line between them */
void plot_cities(gdImagePtr im, Location *city1, Location *city2,
                   int color) {
...
  /* Convert the two locations into screen coordinates (bunch ‘o math) */
...
  /* Draw the cities */
  gdImageString(im,gdFontSmall,...)
  gdImageString(im,gdFontSmall,...)
  gdImageLine(im,ix1,height-iy1,ix2,height-iy2,color);
}
```

Next, we'll wrap a few critical gd functions into Perl. We don't need the entire library so there's not much sense in wrapping the whole thing (it's easy enough to do if you really want to of course). We'll just wrap a couple of functions to illustrate how it can be used (one might also consider using the already existing gd module for Perl as well).

```
%module gd
%{
#include "gd.h"
%}
typedef struct gdImageStruct gdImage;
typedef gdImage * gdImagePtr;

/* Functions to manipulate images. */
gdImagePtr gdImageCreate(int sx, int sy);
int  gdImageColorAllocate(gdImagePtr im, int r, int g, int b);
%inline %{
  /* Shortcut for dumping a file */
  void dump_gif(gdImagePtr im, char *filename) {
    FILE *f;
    f = fopen(filename, "w");
    gdImageGif(im,f);
    fclose(f);
  }
%}
```

We can now slap everything together using a new interface file like this. we'll keep the old graph module name so our existing scripts still work :

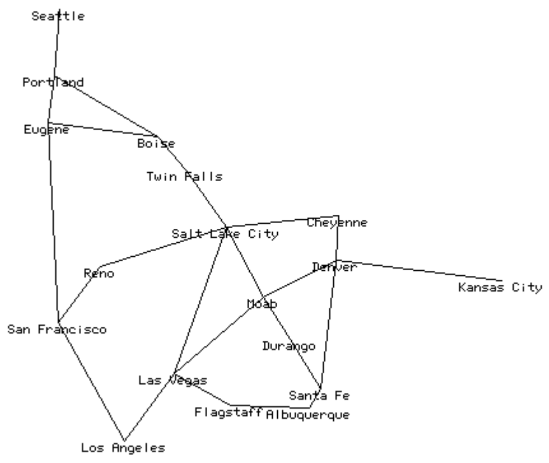
```
// File : package.i
%module graph
%include graph.i           // The original graph program
%include location.i        // The location data structure and functions
%include gd.i              // gd module
%include plot.c            // Function to plot cities
```

Whew! After all of that work, we can do the following :

```
read_cities("cities");
read_locations("locations");
```

```
# Create a new image with gd
$im = gdImageCreate(500,500);
$white = gdImageColorAllocate($im,255,255,255);
$black = gdImageColorAllocate($im,0,0,0);

# Set plotting range (variables in the C code)
$xmin = -130;
$xmax = -90;
$ymin = 30;
$ymax = 50;

# Make a plot of the entire graph
@loc = each %Cities;
while (@loc) {
    my $city = $loc[0];
    my $node = $Cities{$city};
    if (exists $Locations{$node->{v}}) {
       my $loc1 = $Locations{$node->{v}};
       my $e = $node->{adj};
       while (defined($e)) {
             if (exists $Locations{$e->{node}->{v}}) {
                my $loc2 = $Locations{$e->{node}->{v}};
                plot_cities($im,$loc1,$loc2,$black);
             }
             $e = $e->{next};
          }
    }
    @loc = each %Cities;
}
# Dump out a GIF file
dump_gif($im,"test.gif");
```

When run, we now get the following :



Not too bad for just a little work....

Wrapping C libraries and other packages

SWIG can be used to build Perl interfaces to existing C libraries and packages. The general strategy for doing this is as follows :

- Locate important header files for the package.
- Copy them into a SWIG interface file.
- Edit the interface file by removing problematic declarations, unneeded functions, and other clutter (this can often be found by just running SWIG).
- Add support functions to improve the interface.

While SWIG can sometimes be used to simply process a raw header file, the results aren't always what you would expect. By working with a separate interface file, you get an opportunity to clean things up. If you're using a stable package, chances are that it's not going to change suddenly so there is really little problem in doing this. To illustrate the process, we will build a Perl5 interface to MATLAB in the next example.

Building a Perl5 interface to MATLAB

To illustrate the process, we can build a Perl5 interface to MATLAB by wrapping its C API. The MATLAB system consists of the MATLAB engine, and library of functions for creating and manipulating matrices, and a collection of utility function. A full description can be found in the "External Interface Guide" provided with MATLAB. Even if you don't have MATLAB, this section can provide some ideas on how to handle other libraries.

The MATLAB engine interface

The first step in building an interface will be to provide access to the MATLAB engine. This is a separate process that runs in the background, but we need to have some mechanism for starting it and issuing commands. The following functions are defined in the MATLAB interface.,

```
int      engClose(Engine *ep);
int      engEvalString(Engine *ep, char *string);
Matrix  *engGetMatrix(Engine *ep, char *name);
int      engPutMatrix(Engine *ep, Matrix *mp);
Engine  *engOpen(char *startcommand);
void     engOutputBuffer(Engine *ep, char *p, int size);
```

While we could wrap these directly, each function requires an object "Engine". They could be a little annoying to use in Perl since we would have to pass a pointer to the engine with every command. This probably isn't necessary or desired. Thus, we could write some wrappers around these to produce a better interface as follows :

```
// engine.i : SWIG file for MATLAB engine
%{
#define BUFFER_SIZE   32768
static Engine  *eng = 0;
static char   ml_output[BUFFER_SIZE];   /* Result Buffer */
%}

%inline %{
```

```
/* Initialize the MATLAB engine */
int init(void) {
       if (eng) return -1;   /* Already open */
       if (!(eng = engOpen(“matlab42”))) {
         fprintf(stderr,"Unable to open matlab.\n");
         return -1;
       }
       engOutputBuffer(eng,ml_output,BUFFER_SIZE);
       return 0;
}
/* Execute a MATLAB command */
char *matlab(char *c) {
       if (!eng) return “not initialized!”;
       engEvalString(eng, c);
       return &ml_output[0];
}

/* Get a matrix from MATLAB */
Matrix *GetMatrix(char *name) {
       if (!eng) return (Matrix *) 0;
       return(engGetMatrix(eng,name));
}

/* Send a matrix to MATLAB */
int PutMatrix(Matrix *m) {
       if (!eng) return -1;
       return(engPutMatrix(eng,m));
}
%}
```

Wrapping the MATLAB matrix functions

Next, we need to build an interface to the MATLAB matrix manipulation library. This library contains about 30 functions to manipulate both real and complex valued matrices. Here we will only consider real-valued matrices. To provide access to the matrices, we’ll write a different interface file with a list of the functions along with a few helper functions.

```
//
// mx.i : SWIG file for MATLAB matrix manipulation
%inline %{
/* Get an element from a matrix */
double getr(Matrix *mat, int i, int j) {
  double *pr;
  int m;
  pr = mxGetPr(mat);
  m = mxGetM(mat);
  return pr[m*j + i];
}
/* Set an element of a matrix */
void setr(Matrix *mat, int i, int j, double val) {
  double *pr;
  int m;
  pr = mxGetPr(mat);
  m = mxGetM(mat);
    pr[m*j + i] = val;
}
%}

/* Now some MATLAB command */
Matrix *mxCreateFull(int m, int n, int ComplexFlag);
int mxGetM(Matrix *mat);
int mxGetN(Matrix *mat);
char *mxGetName(Matrix *mat);
void mxSetName(Matrix *mat, char *name);
double *mxGetPr(Matrix *mat);
void mxSetPr(Matrix *mat, double *pr);
double mxGetScalar(Matrix *mat);
void mxFreeMatrix(Matrix *pm);
```

Putting it all together

Finally we are ready to put our interface together. There is no need to build a big monolithic interface file--we can simply build it up in pieces :

```
// matlab.i
// Simple SWIG interface to MATLAB
%module matlab
%{
#include “engine.h”
%}
%include engine.i
%include mx.i
```

Our module can be compiled as follows :

```
unix > swig -perl5 matlab.i
unix > gcc -c matlab_wrap.c -I/usr/local/lib/perl5/arch-osname/5.003/CORE -Dbool=char
       -I$(MATLAB)/extern/include
unix > ld -shared matlab_wrap.o -L$(MATLAB)/extern/lib -lmat -o matlab.so
```

Where `$(MATLAB)` is the location of the MATLAB installation (you may have to dig for this).

With our new MATLAB module, we can now write Perl scripts that issue MATLAB commands. For example :

```
use matlab;
matlab::init();
matlab::matlab(“x = -8:.25:8; \
                y = x; \
                [X,Y] = meshgrid(x,y); \
                R = sqrt(X.^2 + Y.^2)+eps; \
                Z = sin(R)./R; \
                mesh(Z); “);
```

Creates a simple 3D surface plot.

Graphical Web-Statistics in Perl5

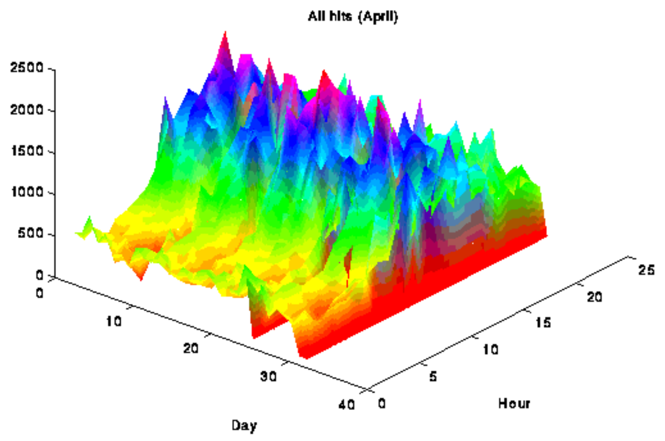
Now, lets use our MATLAB module to generate plots of web-server hits for a given month. To do this, we’ll use our MATLAB module, and create a special purpose function for processing days and hours.

```
// Simple C function for recording a hit
%module webgraph
%inline %
       void hit(double *m, int day, int hour) {
              if ((day >= 0) && (day <= 31)) {
                     *(m+24*(day-1)+hour) += 1.0;
              }
       }
%}
```

While we could write this function in Perl, it will be much faster in C. If we're processing a huge file, then the extra bit of performance will help us out. Once compiled, we can now write a Perl5 script like the following :

```
use matlab;
use webgraph;

# Initialize matlab engine
matlab::init();
# Make a matrix for all hits
$m_all = matlab::mxCreateFull(24,31,0);
matlab::mxSetName($m_all,"all");
$all = matlab::mxGetPr($m_all);         # Get double * of Matrix

foreach $file (@ARGV) {
  open(FILE,$file);
  print "Processing ",$file,"\n";
  while (<FILE>) {
    @fields = split(/\s+/, $_);
    next if ($fields[8] != 200);
    @datetime = split(/\[|\/|:/, $fields[3]);
    if ($datetime[2] =~ /Apr/) {
      webgraph::hit($all, $datetime[1],$datetime[4]);
    }
  }
  # Dump temporary results
}  #foreach
matlab::PutMatrix($m_all);
matlab::matlab("figure(1); \
                surf(all); \
                view(40,40); \
                shading interp; \
                title('All hits'); \
                set(gca,'xlabel',text(0,0,'Day')); \
                set(gca,'ylabel',text(0,0,'Hour')); \
                print -dgif8 stats_all.gif");
```

When run on the web-server logs for the University of Utah, this script produces the following GIF file (showing hits per hour) :



Handling output values (the easy way)

Some C libraries rely on functions that return values in their arguments. For example :

```
void fadd(double a, double b, double *result) {
       *result = a+b;
}
```

By default, SWIG does not handle this case (leaving it up to you to figure out what to do about the `result` parameter). However, this problem is easily fixed using the `typemaps.i` library file (see the Typemaps chapter for more information). For example :

```
// SWIG interface file with output arguments
%module example

%include typemaps.i                          // Load the typemaps librayr
%apply double *OUTPUT {double *result};     // Make "result" an output parameter

void fadd(double a, double b, double *result);
```

When used in a Perl script, the fadd function now takes 2 parameters and returns the result.

```
$a = fadd(3,4);              # $a gets the value of double *result
```

When multiple output arguments are involved, a Perl array of values will be returned instead. It is also possible to use Perl references as function values. This is done as shown :

```
// SWIG interface file with output arguments
%module
%include typemaps.i
```

```
%apply double *REFERENCE {double *result};        // Make “result” a Perl reference

void fadd(double a, double b, double *result);
```

Now in a Perl script, the function works like this :

```
$c = 0.0;                     # Variable for holding the result
fadd(3,4,\$c);                # fadd() places the result in $c
```

In addition to handling pointers as output values or references, two other methods are available. The `INPUT` method indicates that a pointer is an input value and the `BOTH` method indicates that a pointer is both an input and output value. These would be specified as follows :

```
%apply double *INPUT {double *in};         // double *in is an input value
%apply double *BOTH {double *r};           // double *r is both an input/output value
```

This processing of datatypes is built using SWIG’s typemap mechanism. The details of typemaps is described shortly although the `typemaps.i` library already includes a variety of useful typemaps ready for use.

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 similiar 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 delimeter 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 the 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 “perlguts” 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 Conversion Functions

`int SvIV(SV *)`	Convert a Perl scalar value to an integer.
`void sv_setiv(SV *sv, (IV) value)`	Set a Perl scalar to the value of a C integer (in value).
`SV *newSViv((IV) value)`	Create a new Perl scalar from a C value.
`int SvIOK(SV *)`	Checks to see if a Perl scalar is an integer.

Perl Floating Point Conversion Functions

`double SvNV(SV *)`	Convert a Perl scalar value to a double precision float.
`void sv_setnv(SV *, (NV) value)`	Set a Perl5 scalar to the value of a C double.
`SV *newSVnv((NV) value)`	Create a new Perl scalar from a C double.
`int SvNOK(SV *)`	Check to see if a Perl scalar is a floating point value.

Perl String Conversion Functions

`char *SvPV(SV *, int len)`	Convert a scalar value to a char *. Returns the length in len unless you set it to the special value 'na'.
`void sv_setpv(SV *, char *val)`	Copy a NULL terminated ASCII string into a Perl scalar.
`void sv_setpvn(SV *, char *val, int len)`	Copy a string of len bytes into a Perl scalar.
`SV *newSVpv(char *value, int len)`	Create a new Perl scalar value from a char * and length.
`int SvPOK(SV *)`	Checks to see if a Perl scalar is a string.
`void sv_catpv(SV *, char *)`	Appends a string to a scalar value.
`void sv_catpvn(SV *, char *, int)`	Appends a string of specified length to a scalar value.

Perl References

`void sv_setref_pv(SV *, char *, void *ptr)`	Create a blessed reference.
`int sv_isobject(SV *)`	Checks to see if a scalar corresponds to an object (is a reference).
`SV *SvRV(SV *)`	Returns a scalar value from a reference.
`int sv_isa(SV *, char *)`	Checks the type of a reference given a name.

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).

Function argument typemaps

`int,` `short,` `long,`	`%typemap(perl5,in) int,short,long {` `$target = ($type) SvIV($source);` `}`
`float,` `double`	`%typemap(perl5,in) float, double {` `$target = ($type) SvNV($source);` `}`

Function argument typemaps

`char *`	`%typemap(perl5,in) char * {` `$target = SvPV($source,na);` `}`

Function return typemaps

`int,` `short,` `long`	`%typemap(perl5,out) int,short,long {` `$target = sv_newmortal();` `sv_setiv($target,(IV) $source);` `argvi++;` `}`
`float,` `double`	`%typemap(perl5,out) float, double {` `$target = sv_newmortal();` `sv_setiv($target,(double) $source);` `argvi++;` `}`
`char *`	`%typemap(perl5,out) char * {` `$target = sv_newmortal();` `sv_setpv($target,$source);` `argvi++;` `}`

Pointer handling

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

SWIG Pointer Conversion Functions

`void sv_setref_pv(SV *sv, char *type, void *ptr)`	Turn the scalar value sv into a pointer. type is the type string and ptr is the actual pointer value.
`char *SWIG_GetPtr(SV *sv, void **ptr, char *type)`	Extract a pointer value from scalar value sv. Performs type-checking and pointer casting if necessary. If type is NULL, any pointer value will be accepted. The result will be stored in ptr and NULL returned. On failure, a pointer to the invalid portion of the type string will be returned.

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.

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 owership 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 {
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 and Python 9

This chapter describes SWIG's support of Python. Many of the example presented here will have a scientific bias given Python's increasing use in scientific applications (this is how I primarily use Python), but the techniques are widely applicable to other areas.

Preliminaries

SWIG 1.1 works with Python 1.3 and Python 1.4. Given the choice, you should use the latest version of Python. 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

To build a Python module, run SWIG using the `-python` option :

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

This will produce 2 files. The file `example_wrap.c` contains all of the C code needed to build a Python module and a documentation file describes the resulting interface. To build a Python module, you will need to compile the file `example_wrap.c` and link it with the rest of your program (and possibly Python itself). When working with shadow classes, SWIG will also produce a `.py` file, but this is described later.

Getting the right header files

In order to compile, you need to locate the following directories that are part of the Python distribution :

For Python 1.3 :

```
/usr/local/include/Py
/usr/local/lib/python/lib
```

For Python 1.4 :

```
/usr/local/include/python1.4
/usr/local/lib/python1.4/config
```

The exact location may vary on your machine, but the above locations are typical.

Compiling a dynamic module

To build a shared object file, you need to compile your module in a manner similar to the following (shown for Irix):

```
% swig -python example.i
% gcc -c example.c
% gcc -c example_wrap.c -DHAVE_CONFIG_H -I/usr/local/include/python1.4 \
       -I/usr/local/lib/python1.4/config
% ld -shared example.o example_wrap.o -o examplemodule.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 "`examplemodule.so`" (or equivalent depending on your machine). The name of the module is specified using the `%module` directive or `-module` command line option.

While dynamic loading is the preferred method for making SWIG modules, it is not foolproof and not supported on all machines. In these cases, you can rebuild the Python interpreter with your extensions added.

Rebuilding the Python interpreter (aka. static linking)

The normal 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. While it's possible to simplify the process by using the VPATH feature of 'make', I've always found the process to be a little too complicated.

SWIG provides an extremely easy, although somewhat unconventional, mechanism for rebuilding Python using SWIG's library feature. When you want to build a static version of Python, simply make an interface file like this :

```
%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 build your module, simply do the following :

```
% swig -python example.i
% gcc example.c example_wrap.c -DHAVE_CONFIG_H -I/usr/local/include/python1.4 \
       -I/usr/local/lib/python1.4/config \
       -L/usr/local/lib/python1.4/config -lModules -lPython -lObjects -lParser -lm \
       -o mypython
```

On some machines, you may need need to supply additional libraries on the link line. In particular, you may need to supply `-lsocket`, `-lnsl`, and `-ldl`.

It is also possible to add the embed.i library to an existing interface by running SWIG as follows :

```
% swig -python -lembed.i example.i
```

The `embed.i` file uses all of the modules that are currently being used in your installed version of Python. Thus, your new version of Python will be identical to the old one except with your new module added. If you have configured Python to use modules such as `tkinter,` you may need to supply linkage to the Tcl/Tk libraries and X11 libraries.

Python's `main()` program is rather unfriendly towards C++ code, but SWIG's `embed.i` module provides a replacement that can be compiled with the C++ compiler--making it easy to build C++ Python extensions.

The `embed.i` library should only be used with Python 1.4. If you are using Python 1.3, you should use the file `embed13.i` instead (this can be done by making a symbolic link in the SWIG library) or simply using the `-l` option.

Using your module

To use your module in Python, simply use Python's import command. The process is identical regardless of whether or not you used dynamic loading or rebuilt the Python interpreter :

```
% python
>>> import example
>>> example.fact(4)
24
>>>
```

Compilation problems and compiling with C++

For the most part, compiling a Python module is straightforward, but there are a number of potential problems :

- Dynamic loading is not supported on all machines. If you can't get a module to build, you might try building a new version of Python using static linking instead.
- 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 examplemodule.so`'
- If building a dynamic C++ module using g++, you may also need to link against `libgcc.a`, `libg++.a`, and `libstc++.a` libraries.
- Make sure you are using the correct header files and libraries. A module compiled with Python 1.3 headers probably won't work with Python 1.4.

Building Python Extensions under Windows 95/NT

Building a SWIG extension to Python 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 Python interpreter. This section covers the process of using SWIG with Microsoft Visual C++ 4.x although the procedure may be similar with other compilers. SWIG currently supports both the basic Python release and Pythonwin. In order to build extensions, you will need to download the source distribution to these packages as you will need the Python 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 -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 "python14.lib". Also, set the name of the output file to match the name of your Python 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 Python extension, simply run Python and use the `import` command as normal. For example :

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

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 Python 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
PY_INCLUDE   = -Id:\python-1.4\Include -Id:\python-1.4 -Id:\python-1.4\Pc
PY_LIB       = d:\python-1.4\vc40\python14.lib
PY_FLAGS = /D__WIN32__

python::
       swig -python -o $(WRAPFILE) $(INTERFACE)
       $(CC) $(CFLAGS) $(PY_FLAGS) $(PY_INCLUDE) $(SRCS) $(WRAPFILE)
       set LIB=$(TOOLS)\lib
       $(LINK) $(LOPT) -out:example.dll $(LIBS) $(PY_LIB) example.obj example_wrap.obj
```

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 Python/C interface

The SWIG Python 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 often used to create more sophisticated interfaces (such as shadow classes) so it may be hidden in practice.

Modules

The SWIG `%module` directive specifies the name of the Python module. If you specified '`%module example`', then everything found in a SWIG interface file will be contained within the Python '`example`' module. Make sure you don't use the same name as a built-in Python command or standard module or your results may be unpredictable.

Functions

C/C++ functions are mapped directly into a matching Python function. For example :

```
%module example
```

```
extern int fact(int n);
```

gets turned into the Python function `example.fact(n)` :

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

Variable Linking

SWIG provides access to C/C++ global variables, but the mechanism is slightly different than one might expect due to the object model used in Python. 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. In other words, there is only one object containing 3.4 and "a" and "b" are both names that refer to it. This is a very different model than that used in C. For this reason, there is no mechanism for mapping "assignment" in Python onto C global variables (because assignment is Python is really a naming operation).

To provide access to C global variables, SWIG creates a special Python object called '`cvar`' that is added to each SWIG generated module. This object is used to access global variables as follows :

```
// SWIG interface file with global variables
%module example
...
extern int My_variable;
extern double density;
...
```

Now in Python :

```
>>> 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
```

Just remember, all C globals need to be prefixed with a "`cvar.`" and you will be set. 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
```

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 will only be able to access global variables from the last module loaded. SWIG does not create cvar if there are no global variables in a module.

Constants

C/C++ constants are installed as new Python objects containing the appropriate value. These constants are given the same name as the corresponding C constant. "Constants" are not guaranteed to be constants in Python---in other words, you are free to change them and suffer the consequences!

Pointers

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

```
_100f8e2_Vector_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_Vector_p
```

To some Python users, the idea of representing pointers as strings may seem strange, but keep in mind that pointers are meant to be opaque objects. In practice, you may never notice that pointers are character strings. There is also a certain efficiency in using this representation as it is easy to pass pointers around between modules and it is unnecessary to rely on a new Python datatype. Eventually, pointers may be represented as special Python objects, but the string representation works remarkably well so there has been little need to replace it.

Structures

The low-level SWIG interface only provides a simple 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)
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 Python 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
>>> 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.

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 will be 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 Python, these functions used to access the C++ class :

```
>>> 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
_1008560_List_p
>>>
```

C++ objects are really just pointers. 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 C++ objects. As it turns out, a more elegant method of accessing structures and classes is available using shadow classes.

Python 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 Python. However, it is possible to use the low-level C interface to write a Python class that looks like the original C++ class. In this case, the Python 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.

A simple example

For our earlier List class, a Python shadow class could be written by hand like this :

```
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
```

When used in a Python script, we can use the class as follows :

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

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

Why write shadow classes in Python?

While one could wrap C/C++ objects directly into Python as new Python 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. It also becomes hard to handle inheritance and more advanced language features. A second, and more serious problem, is that Python "types" created in C can not be subclassed or used in the same way as one might use a real Python class. As a result, it is not possible to do interesting things like create Python classes that inherit from C++ classes.

By writing shadow classes in Python instead of C, the classes become real Python classes that can be used as base-classes in an inheritance hierarchy or for other applications. Writing the shadow classes in Python also greatly simplies coding complexity as writing in Python is much easier than trying to accomplish the same thing in C. Finally, by writing shadow classes in Python, they are easy to modify and can be changed without ever recompiling any of the C code. The down-side to this approach is worse performance--a concern for some users.

The problems of combining C++ and Python have been of great interest to the Python community. SWIG is primarily concerned with accessing C++ from Python. Readers who are interested in more than this (and the idea of accessing Python classes from C++) are encouraged to look into the MESS extension which aims to provide a tighter integration between C++ and Python. The recently announced GRAD package also shows much promise and provides very comprehensive C++/Python interface.

Automated shadow class generation

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

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

This will create the following two files :

```
interface_wrap.c
module.py
```

The file `interface_wrap.c` contains the normal SWIG C/C++ wrappers. The file `module.py` contains the Python code corresponding to shadow classes. The name of this file will be the same as specified by the `%module` directive in the SWIG interface file.

Associated with the two files are TWO Python modules. The C module '`modulec`' contains the low-level C interface that would have been created without the `-shadow` option. The Python module '`module`' contains the Python shadow classes that have been built around the low-level interface. To use the module, simply use '`import module`'. For all practical purposes, the '`modulec`' module is completely hidden although you can certainly use it if you want to.

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.

When using static linking, no changes need to be made to the compilation process.

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 and is not necessary to effectively use SWIG.

About the Examples

The next few sections will go through a series of Python examples of varying complexity. These examples are designed to illustrate how SWIG can be used to integrate C/C++ and Python in a variety of ways. Some of the things that will be covered include :

- Controlling a simple C++ program with Python
- Wrapping a C library.
- Adding Python methods to existing C++ classes
- Accessing arrays and other common data structures.
- Building reusable components.
- Writing C/C++ callback functions in Python.

Solving a simple heat-equation

In this example, we will show how Python can be used to control a simple physics application--in this case, some C++ code for solving a 2D heat equation. This example is probably overly simplistic, but hopefully it's enough to give you some ideas.

The C++ code

Our simple application consists of the following two files :

```
// File : pde.h
// Header file for Heat equation solver

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

// A simple 2D Grid structure

// A simple structure for holding a 2D grid of values
struct Grid2d {
  Grid2d(int ni, int nj);
  ~Grid2d();
  double **data;
  int      xpoints;
  int      ypoints;
};
```

```
// Simple class for solving a heat equation */
class Heat2d {
private:
  Grid2d    *work;                        // Temporary grid, needed for solver
  double    h,k;                          // Grid spacing
public:
  Heat2d(int ni, int nj);
  ~Heat2d();
  Grid2d    *grid;                        // Data
  double     dt;                          // Timestep
  double     time;                        // Elapsed time
  void       solve(int nsteps);           // Run for nsteps
  void       set_temp(double temp);       // Set temperature
};
```

The supporting C++ code implements a simple partial differential equation solver and some operations on the grid data structure. The precise implementation isn't important here, but all of the code can be found in the "`Examples/python/manual`" directory of the SWIG distribution.

Making a quick and dirty Python module

Given our simple application, making a Python module is easy. Simply use the following SWIG interface file :

```
// File : pde.i
%module pde
%{
#include "pde.h"
%}

%include pde.h
```

Since `pde.h` is fairly simple, we can simply include it directly into our interface file using `%include`. However, we also need to make sure we also include it in the `%{,%}` block--otherwise we'll get a huge number of compiler errors when we compile the resulting wrapper file.

To build the module simply run SWIG with the following options

```
swig -python -shadow pde.i
```

and compile using the techniques described in the beginning of this chapter.

Using our new module

We are now ready to use our new module. To do this, we can simply write a Python script like this :

```
# A fairly uninteresting example

from pde import *

h = Heat2d(50,50)           # Creates a new "problem"
```

```
h.set_temp(1.0)
print "Dt = ", h.dt

# Solve something

for i in range(0,25):
       h.solve(100)
       print "time = ", h.time
```

When run, we get rather exciting output such as the following :

```
Dt =  2.5e-05
time =  0.0025
time =  0.005
time =  0.0075
...
time =  0.06
time =  0.0625
```

(okay, it's not that exciting--well, maybe it is if you don't get out much).

While this has only been a simple example it is important to note that we could have just as easily written the same thing in C++. For example :

```
// Python example written in C++

#include "pde.h"
#include <stdio.h>

int main(int argc, char **argv) {

  Heat2d *h;

  h = new Heat2d(50,50);
  printf("Dt = %g\n", h->dt);

  h->set_temp(1.0);

  for (int i = 0; i < 25; i++) {
    h->solve(100);
    printf("time = %g\n", h->time);
  }
}
```

For the most part, the code looks identical (although the Python version is simpler). As for performance, the Python version runs less than 1% slower than the C++ version on my machine. Given that most of the computational work is written in C++, there is very little performance penalty for writing the outer loop of our calculation in Python in this case.

Unfortunately, our Python version suffers a number of drawbacks. Most notably, there is no way for us to access any of the grid data (which is easily accomplished in C++). However, there are ways to fix this :

Accessing array data

Let’s modify our heat equation problem so that we can access grid data directly from Python. This can be done by modifying our interface file as follows :

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

%include pde.h

// Add a few "helper" functions to extract grid data
%inline %{
double  Grid2d_get(Grid2d *g, int i, int j) {
      return g->data[i][j];
}
void    Grid2d_set(Grid2d *g, int i, int j, double val) {
      g->data[i][j] = val;
}
%}
```

Rather than modifying our C++ code, it is easy enough to supply a few accessor functions directly in our interface file. These function may only be used from Python so this approach makes sense and it helps us keep our C++ code free from unnecessary clutter. The `%inline` directive is a convenient method for adding helper functions since the functions you declare show up in the interface automatically.

We can now use our accessor functions to write a more sophisticated Python script :

```
# An example using our set/get functions

from pde import *

# Set up an initial condition
def initcond(h):
       h.set_temp(0.0)
       nx = h.grid.xpoints
       for i in range(0,nx):
              Grid2d_set(h.grid,i,0,1.0)                       # Set grid values

# Dump out to a file
def dump(h,filename):
       f = open(filename,"w")
       nx = h.grid.xpoints
       ny = h.grid.ypoints
       for i in range(0,nx):
              for j in range(0,ny):
                     f.write(str(Grid2d_get(h.grid,i,j))+"\n")  # Get grid value
       f.close()

# Set up a problem and run it

h = Heat2d(50,50)
initcond(h)
fileno = 1
for i in range(0,25):
```

```
                h.solve(100)
                dump(h,"Dat"+str(fileno))
                print "time = ", h.time
                fileno = fileno+1
```

We now have a Python script that can create a grid, set up an initial condition, run a simulation, and dump a collection of datafiles. So, with just a little supporting code in our interface file, we can start to do useful work from Python.

Use Python for control, C for performance

Now that it is possible to access grid data from Python, it is possible to quickly write code for all sorts of operations. However, Python may not provide enough performance for certain operations. For example, the `dump()` function in the previous example may become quite slow as problem sizes increase. Thus, we might consider writing it in C++ such as the follows:

```
void dump(Heat2d *h, char *filename) {
      FILE *f;
      int   i,j;

      f = fopen(filename,"w");
      for (i = 0; i < h->grid->xpoints; i++)
       for (j = 0; j < h->grid->ypoints; j++)
         fprintf(f,"%0.17f\n",h->grid->data[i][j]);
      fclose(f);
}
```

To use this new function, simple put its declaration in the SWIG interface file and get rid of the old Python version. The Python script won't know that you changed the implementation.

Getting even more serious about array access

We have provided access to grid data using a pair of get/set functions. However, using these functions is a little clumsy because they always have to be called as a separate function like this :

```
Grid2d_set(grid,i,j,1.0)
```

It might make more sense to make the get/set functions appear like member functions of the `Grid2D` class. That way we could use them like this :

```
grid.set(i,j,1.0)
grid.get(i,j)
```

SWIG provides a simple technique for doing this as illustrated in the following interface file :

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

// Add a few "helper" functions to extract grid data
%{
    double  Grid2d_get(Grid2d *g, int i, int j) {
          return g->data[i][j];
        }
        void    Grid2d_set(Grid2d *g, int i, int j, double val) {
          g->data[i][j] = val;
        }
%}

// Now add these helper functions as methods of Grid2d

%addmethods Grid2d {
    double get(int i, int j);              // Gets expanded to Grid2d_get()
    void   set(int i, int j, double val);  // Gets expanded to Grid2d_set()
}
```

The `%addmethods` directive tells SWIG that you want to add new functions to an existing C++ class or C structure for the purposes of building an interface. In reality, SWIG leaves the original C++ class unchanged, but the resulting Python interface will have some new functions that appear to be class members.

SWIG uses a naming convention for adding methods to a class. If you have a class `Foo` and you add a member function `bar(args)`, SWIG will look for a function called `Foo_bar(this,args)` that implements the desired functionality. You can write this function yourself, as in the previous interface file, but you can also just supply the code immediately after a declaration like this :

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

// Add some new accessor methods to the Grid2D class
%addmethods Grid2d {
  double get(int i, int j) {
    return self->data[i][j];
  };
  void set(int i, int j, double val) {
    self->data[i][j] = val;
  };
};
```

In this case, SWIG will take the supplied code, and automatically generate a function for the method. The special variable "`self`" is used to hold a pointer to the corresponding object. The `self` pointer is exactly like the C++ "`this`" pointer, except that the name has been changed in order to remind you that you aren't really writing a real class member function. (Actually, the real reason we can't use "this" is because the C++ compiler will start complaining!)

Finally, it is worth noting that the `%addmethods` directive may also be used inside a class definition like this :

```
struct Grid2d {
  Grid2d(int ni, int nj);
  ~Grid2d();
  double **data;
```

```
    int      xpoints;
    int      ypoints;
    %addmethods {
         double get(int i, int j);
         void   set(int i, int j, double value);
    }
  };
```

This latter case is really only useful if the C++ class definition is included in the SWIG interface file itself. If you are pulling the class definition out of a separate file or a C++ header file, using a separate `%addmethods` directive is preferable. It doesn't matter if the `%addmethods` directive appears before or after the real class definition--SWIG will correctly associate the two definitions.

Okay, enough talk. By adding the set/get functions as methods, we can now change our Python script to look like this (changes are underlined) :

```
# An example using our new set/get functions

from pde import *

# Set up an initial condition

def initcond(h):
       h.set_temp(0.0)
       nx = h.grid.xpoints
       for i in range(0,nx):
              h.grid.set(i,0,1.0)         # Note changed interface

# Dump out to a file
def dump(h,filename):
       f = open(filename,"w")
       nx = h.grid.xpoints
       ny = h.grid.ypoints
       for i in range(0,nx):
              for j in range(0,ny):
                     f.write(str(h.grid.get(i,j))+"\n")
       f.close()

# Set up a problem and run it

h = Heat2d(50,50)
initcond(h)
fileno = 1

for i in range(0,25):
       h.solve(100)
       h.dump("Dat"+str(fileno))
       print "time = ", h.time
       fileno = fileno+1
```

Now it's starting to look a little better, but we can do even better...

Implementing special Python methods in C

Now that you're getting into the spirit of things, let's make it so that we can access our `Grid2D` data like a Python array. As it turns out, we can do this with a little trickery in the SWIG interface file. Don't forget to put on your Python wizard cap...

```
// SWIG interface file with Python array methods added
%module pde
%{
#include "pde.h"
%}

%include pde.h

%inline %{
  // Define a new Grid2d row class
  struct Grid2dRow {
    Grid2d *g;       // Grid
    int    row;      // Row number
    // These functions are used by Python to access sequence types (lists, tuples, ...)
    double __getitem__(int i) {
      return g->data[row][i];
    };
    void __setitem__(int i, double val) {
      g->data[row][i] = val;
    };
  };
%}

// Now add a __getitem__ method to Grid2D to return a row
%addmethods Grid2d {
  Grid2dRow __getitem__(int i) {
    Grid2dRow r;
    r.g = self;
    r.row = i;
    return r;
  };
};
```

We have now replaced our get/set functions with `the __getitem__` and `__setitem__` functions that Python needs to access arrays. We have also added a special `Grid2dRow` class. This is needed to allow us to make a funny kind of "multidimensional" array in Python (this may take a few minutes of thought to figure out). Using this new interface file, we can now write a Python script like this :

```
# An example script using our array access functions

from pde import *

# Set up an initial condition

def initcond(h):
       h.set_temp(0.0)
       nx = h.grid.xpoints
       for i in range(0,nx):
              h.grid[i][0] = 1.0            # Note nice array access

# Set up a problem and run it

h = Heat2d(50,50)
initcond(h)
```

```
fileno = 1

for i in range(0,25):
       h.solve(100)
       h.dump("Dat"+str(fileno))
       print "time = ", h.time
       fileno = fileno+1

# Calculate average temperature over the region

sum = 0.0
for i in range(0,h.grid.xpoints):
       for j in range(0,h.grid.ypoints):
              sum = sum + h.grid[i][j]        # Note nice array access

avg = sum/(h.grid.xpoints*h.grid.ypoints)

print "Avg temperature = ",avg
```

Summary (so far)

In our first example, we have taken a very simple C++ problem and wrapped it into a Python module. With a little extra work, we have been able to provide array type access to our C++ data from Python and to write some computationally intensive operations in C++. At this point, it would easy to write all sorts of Python scripts to set up problems, run simulations, look at the data, and to debug new operations implemented in C++.

Wrapping a C library

In this next example, we focus on wrapping the gd-1.2 library. gd is public domain library for fast GIF image creation written by Thomas Boutell and available on the internet. gd-1.2 is copyright 1994,1995, Quest Protein Database Center, Cold Spring Harbor Labs. This example assumes that you have gd-1.2 available, but you can use the ideas here to wrap other kinds of C libraries.

Preparing a module

Wrapping a C library into a Python module usually involves working with the C header files associated with a particular library. In some cases, a header file can be used directly (without modification) with SWIG. Other times, it may be necessary to copy the header file into a SWIG interface file and make a few touch-ups and modifications. In either case, it's usually not too difficult.

To make a module, you can use the following checklist :

- Locate the header files associated with a package
- Look at the contents of the header files to see if SWIG can handle them. In particular, SWIG can not handle excessive use of C preprocessor macros, or non-ANSI C syntax. The best way to identify problems is to simply run SWIG on the file and see what errors (if any) get reported.
- Make a SWIG interface file for your module specifying the name of the module, the appropriate header files, and any supporting documentation that you would like to provide.

- If the header file is clean, simply use SWIG's `%include` directive. If not, paste the header file into your interface file and edit it until SWIG can handle it.
- Clean up the interface by possibly adding supporting code, deleting unnecessary functions, and eliminating clutter.
- Run SWIG and compile.

In the case of the gd library, we can simply use the following SWIG interface file :

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

%section "gd-1.2",ignore
%include "gd.h"

// These will come in handy later
FILE *fopen(char *, char *);
void fclose(FILE *f);
```

In this file, we first tell SWIG to put all of the gd functions in a separate documentation section and to ignore all comments. This usually helps clean up the documentation when working with raw header files. Next, we simply include the contents of "gd.h" directly. Finally, we provide wrappers to `fopen()` and `fclose()` since these will come in handy in our Python interface.

If we give this interface file to SWIG, we will get the following output :

```
% swig -python -shadow  -I/usr/local/include gd.i
Generating wrappers for Python
/usr/local/include/gd.h : Line 32.  Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 33.  Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 34.  Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 35.  Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 41.  Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 42.  Arrays not currently supported (ignored).
%
```

While SWIG was able to handle most of the header file, it also ran into a few unsupported declarations---in this case, a few data structures with array members. However, the warning messages also tell us that these declarations have simply been ignored. Thus, we can choose to continue and build our interface anyways. As it turns out in this case, the ignored declarations are of little or no consequence so we can ignore the warnings.

If SWIG is unable to process a raw header file or if you would like to eliminate the warning messages, you can structure your interface file as follows :

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

%section "gd-1.2",ignore
```

```
... paste the contents of gd.h here and remove problems ...

// A few extra support functions

FILE *fopen(char *, char *);
void fclose(FILE *f);
```

This latter option requires a little more work (since you need to paste the contents of gd.h into the file and edit it), but is otherwise not much more difficult to do. For highly complex C libraries or header files that go overboard with the C preprocessor, you may need to do this more often.

Using the gd module

Now, that we have created a module from the gd library, we can use it in Python scripts. The following script makes a simple image of a black background with a white line drawn on it. Notice how we have used our wrapped versions of `fopen()` and `fclose()` to create a FILE handle for use in the gd library (there are also ways to use Python file objects, but this is described later).

```
# Simple gd program

from gd import *

im = gdImageCreate(64,64)
black = gdImageColorAllocate(im,0,0,0)
white = gdImageColorAllocate(im,255,255,255)
gdImageLine(im,0,0,63,63,white)
out = fopen("test.gif","w")
gdImageGif(im,out)
fclose(out)
gdImageDestroy(im)
```

That was simple enough--and it only required about 5 minutes of work. Unfortunately, our gd module still has a few problems...

Extending and fixing the gd module

While our first attempt at wrapping gd works for simple functions, there are a number of problems. For example, the gd-1.2 library contains the following function for drawing polygons :

```
void gdImagePolygon(gdImagePtr im, gdPointPtr points, int pointsTotal, int color);
```

The `gdImagePtr` type is created by another function in our module and the parameters `pointsTotal` and `color` are simple integers. However, the 2nd argument is a pointer to an array of points as defined by the following data structure in the gd-1.2 header file :

```
typedef struct {
       int x, y;
} gdPoint, *gdPointPtr;
```

Unfortunately, there is no way to create a gdPoint in Python and consequently no way to call the gdImagePolygon function. A temporary setback, but one that is not difficult to solve using the `%addmethods` directive as follows :

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

%include "gd.h"

// Fix up the gdPoint structure a little bit
%addmethods gdPoint {
  // Constructor to make an array of “Points”
  gdPoint(int npoints) {
    return (gdPoint *) malloc(npoints*sizeof(gdPoint));
  };
  // Destructor to destroy this array
  ~gdPoint() {
    free(self);
  };
  // Python method for array access
  gdPoint *__getitem__(int i) {
    return self+i;
  };
};

FILE *fopen(char *, char *);
void fclose(FILE *f);
```

With these simple additions, we can now create arrays of points and use the polygon function as follows :

```
# Simple gd program

from gd import *

im = gdImageCreate(64,64)
black = gdImageColorAllocate(im,0,0,0)
white = gdImageColorAllocate(im,255,255,255)

pts = gdPoint(3);                   # Create an array of Points
pts[0].x,pts[0].y = (5,5)           # Assign a set of points
pts[1].x,pts[1].y = (60,25)
pts[2].x,pts[2].y = (16,60)

gdImagePolygon(im,pts,3,white)      # Draw a polygon from our array of points
out = fopen("test.gif","w")
gdImageGif(im,out)
fclose(out)
gdImageDestroy(im)
```

Building a simple 2D imaging class

Now it’s time to get down to business. Using our gd-1.2 module, we can write a simple 2D imaging class that hides alot of the underlying details and provides scaling, translations, and a host of other operations. (It’s a fair amount code, but an interesting example of how one can take a simple C library and turn it into something that looks completely different).

```
# image.py
# Generic 2D Image Class
#
# Built using the 'gd-1.2' library by Thomas Boutell
#

import gd

class Image:
       def __init__(self,width,height,xmin=0.0,ymin=0.0,xmax=1.0,ymax=1.0):
              self.im = gd.gdImageCreate(width,height)
              self.xmin   = xmin
              self.ymin   = ymin
              self.xmax   = xmax
              self.ymax   = ymax
              self.width  = width
              self.height = height
              self.dx     = 1.0*(xmax-xmin)
              self.dy     = 1.0*(ymax-ymin)
              self.xtick  = self.dx/10.0
              self.ytick  = self.dy/10.0
              self.ticklen= 3
              self.name   = "image.gif"
              gd.gdImageColorAllocate(self.im,0,0,0)        # Black
              gd.gdImageColorAllocate(self.im,255,255,255)  # White
              gd.gdImageColorAllocate(self.im,255,0,0)      # Red
              gd.gdImageColorAllocate(self.im,0,255,0)      # Green
              gd.gdImageColorAllocate(self.im,0,0,255)      # Blue

       def __del__(self):
              print "Deleting"
              gd.gdImageDestroy(self.im)

       # Dump out this image to a file
       def write(self,name="NONE"):
              if name == "NONE":
                     name = self.name
              f = gd.fopen(name,"w")
              gd.gdImageGif(self.im,f)
              gd.fclose(f)
              self.name = name

       # Virtual method that derived classes define
       def draw(self):
              print "No drawing method specified."

       # A combination of write and draw
       def show(self,filename="NONE"):
              self.draw()
              self.write(filename)

       # Load up a colormap from a Python array of (R,G,B) tuples
       def colormap(self, cmap):
              for i in range(0,255):
                     gd.gdImageColorDeallocate(self.im,i)
              for c in cmap:
                     gd.gdImageColorAllocate(self.im,c[0],c[1],c[2])

       # Change viewing region
```

```
def region(self,xmin,ymin,xmax,ymax):
       self.xmin = xmin
       self.ymin = ymin
       self.xmax = xmax
       self.ymax = ymax
       self.dx     = 1.0*(xmax-xmin)
       self.dy     = 1.0*(ymax-ymin)

# Transforms a 2D point into screen coordinates
def transform(self,x,y):
       npt = []
       ix = (x-self.xmin)/self.dx*self.width + 0.5
       iy = (self.ymax-y)/self.dy*self.height + 0.5
       return (ix,iy)

# A few graphics primitives
def clear(self,color):
       gd.gdImageFilledRectangle(self.im,0,0,self.width,self.height,color)

def plot(self,x,y,color):
       ix,iy = self.transform(x,y)
       gd.gdImageSetPixel(self.im,ix,iy,color)

def line(self,x1,y1,x2,y2,color):
       ix1,iy1 = self.transform(x1,y1)
       ix2,iy2 = self.transform(x2,y2)
       gd.gdImageLine(self.im,ix1,iy1,ix2,iy2,color)

def box(self,x1,y1,x2,y2,color):
       ix1,iy1 = self.transform(x1,y1)
       ix2,iy2 = self.transform(x2,y2)
       gd.gdImageRectangle(self.im,ix1,iy1,ix2,iy2,color)

def solidbox(self,x1,y1,x2,y2,color):
       ix1,iy1 = self.transform(x1,y1)
       ix2,iy2 = self.transform(x2,y2)
       gd.gdImageFilledRectangle(self.im,ix1,iy1,ix2,iy2,color)

def arc(self,cx,cy,w,h,s,e,color):
       ix,iy = self.transform(cx,cy)
       iw = (x - self.xmin)/self.dx * self.width
       ih = (y - self.ymin)/self.dy * self.height
       gd.gdImageArc(self.im,ix,iy,iw,ih,s,e,color)

def fill(self,x,y,color):
       ix,iy = self.transform(x,y)
       gd.gdImageFill(self,ix,iy,color)

def axis(self,color):
       self.line(self.xmin,0,self.xmax,0,color)
       self.line(0,self.ymin,0,self.ymax,color)
       x = -self.xtick*(int(-self.xmin/self.xtick)+1)
       while x <= self.xmax:
           ix,iy = self.transform(x,0)
           gd.gdImageLine(self.im,ix,iy-self.ticklen,ix,iy+self.ticklen,color)
           x = x + self.xtick
       y = -self.ytick*(int(-self.ymin/self.ytick)+1)
       while y <= self.ymax:
           ix,iy = self.transform(0,y)
```

```
                        gd.gdImageLine(self.im,ix-self.ticklen,iy,ix+self.ticklen,iy,color)
                        y = y + self.ytick

        # scalex(s).  Scales the x-axis.  s is given as a scaling factor
        def scalex(self,s):
                xc = self.xmin + self.dx/2.0
                dx = self.dx*s
                xmin = xc - dx/2.0
                xmax = xc + dx/2.0
                self.region(xmin,self.ymin,xmax,self.ymax)

        # scaley(s).  Scales the y-axis.
        def scaley(self,s):
                yc = self.ymin + self.dy/2.0
                dy = self.dy*s
                ymin = yc - dy/2.0
                ymax = yc + dy/2.0
                self.region(self.xmin,ymin,self.xmax,ymax)

        # Zooms a current image.  s is given as a percent
        def zoom(self,s):
                s = 100.0/s
                self.scalex(s)
                self.scaley(s)

        # Move image left.  s is given in range 0,100. 100 moves a full screen left
        def left(self,s):
                dx = self.dx*s/100.0
                xmin = self.xmin + dx
                xmax = self.xmax + dx
                self.region(xmin,self.ymin,xmax,self.ymax)

        # Move image right.  s is given in range 0,100. 100 moves a full screen right
        def right(self,s):
                self.left(-s)

        # Move image down.  s is given in range 0,100. 100 moves a full screen down
        def down(self,s):
                dy = self.dy*s/100.0
                ymin = self.ymin + dy
                ymax = self.ymax + dy
                self.region(self.xmin,ymin,self.xmax,ymax)

        # Move image up.  s is given in range 0,100. 100 moves a full screen up
        def up(self,s):
                self.down(-s)

        # Center image
        def center(self,x,y):
                self.right(50-x)
                self.up(50-y)
```

Our image class provides a number of methods for creating images, plotting points, making lines, and other graphical objects. We have also provided some methods for moving and scaling the image. Now, let’s use this image class to do some interesting things :

A mathematical function plotter

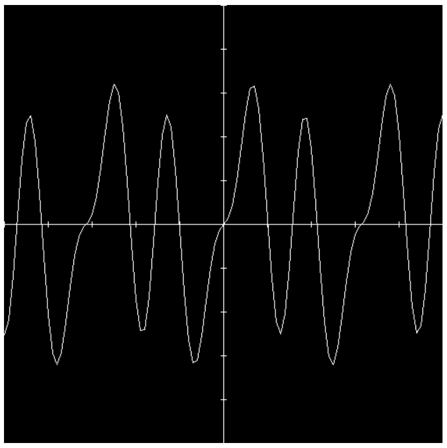
Here’s a simple class that can be used to plot mathematical functions :

```
# funcplot.py

from image import *

class PlotFunc(Image):
	def __init__(self,func,xmin,ymin,xmax,ymax,width=500,height=500):
		Image.__init__(self,width,height,xmin,ymin,xmax,ymax)
		self.func = func		# The function being plotted
		self.npoints = 100		# Number of samples
		self.color = 1
	def draw(self):
		self.clear(0)
		lastx = self.xmin
		lasty = self.func(lastx)
		dx = 1.0*(self.xmax-self.xmin)/self.npoints
		x = lastx+dx
		for i in range(0,self.npoints):
			y = self.func(x)
			self.line(lastx,lasty,x,y,self.color)
			lastx = x
			lasty = y
			x = x + dx
		self.axis(1)
```

Most of the functionality is implemented in our base image class so this is pretty simple. However, if we wanted to make a GIF image of a mathematical function, we could just do this :

```
>>> from funcplot import *
>>> import math
>>> p = PlotFunc(lambda x: 0.5*math.sin(x)+0.75*math.sin(2*x)-0.6*math.sin(3*x),
                 -10,-2,10,2)
>>> p.show(“plot.gif”)
```

Which would produce the following GIF image :



Plotting an unstructured mesh

Of course, perhaps we want to plot something a little more complicated like a mesh. Recently, a colleague came to me with some unstructured mesh data contained in a pair of ASCII formatted files. These files contained a collection of points, and a list of connectivities defining a mesh on these points. Reading and plotting this data in Python turned out to be relatively easy using the following script and our image base class :

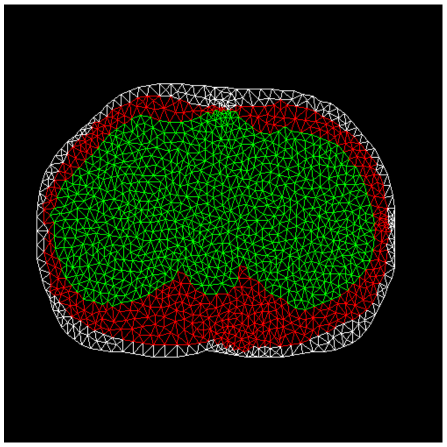
```
# plotmesh.py
# Plots an unstructured mesh stored as an ASCII file
from image import *
import string

class PlotMesh(Image):
        def __init__(self,filename,xmin,ymin,xmax,ymax,width=500,height=500):
                Image.__init__(self,width,height,xmin,ymin,xmax,ymax)
                # read in a mesh file in pieces
                pts = []
                # Read in data points
                atoi = string.atoi
                atof = string.atof
                f = open(filename+".pts","r")
                npoints = atoi(f.readline())
                for i in range(0,npoints):
                        l = string.split(f.readline())
                        pts.append((atof(l[0]),atof(l[1])))
                f.close()

                # Read in mesh data
                f = open(filename+".tris","r")
                ntris = string.atoi(f.readline())
                tris = [ ]
                for i in range(0,ntris):
                        l = string.split(f.readline())
                        tris.append((atoi(l[0])-1,atoi(l[1])-1,atoi(l[2])-1,atoi(l[3])))
                f.close()

        # Set up local attributes
                self.pts = pts
                self.npoints = npoints
                self.tris = tris
                self.ntris = ntris

        # Draw mesh
        def draw(self):
                self.clear(0);
                i = 0
                while i < self.ntris:
                        tri = self.tris[i]
                        pt1 = self.pts[tri[0]]
                        pt2 = self.pts[tri[1]]
                        pt3 = self.pts[tri[2]]
                        # Now draw the mesh
                        self.triangle(pt1[0],pt1[1],pt2[0],pt2[1],pt3[0],pt3[1],tri[3]);
                        i = i + 1

        # Draw a triangle
        def triangle(self,x1,y1,x2,y2,x3,y3,color):
```

```
                        self.line(x1,y1,x2,y2,color)
                        self.line(x2,y2,x3,y3,color)
                        self.line(x3,y3,x1,y1,color)
```

This class simply reads the data into a few Python lists, has a drawing function for making a plot, and adds a special method for making triangles. Making a plot is now easy, just do this :

```
>>> from plotmesh.py import *
>>> mesh = PlotMesh("mesh",5,0,35,25)
>>> mesh.show("mesh.gif")
```

This produces the following GIF image :



When run interactively, we can also use simple commands to zoom in and move the image around. For example :

```
>>> mesh = PlotMesh("mesh",5,0,35,25)
>>> mesh.zoom(200)                     # Enlarge by 200%
>>> mesh.left(50)                      # Move image half a screen to left
>>> mesh.show()
>>>
```

While a Python-only implementation would be unsuitable for huge datasets, performance critical operations could be moved to C and used in conjunction with our Image base class.

From C to SWIG to Python

This example illustrates a number of things that are possible with SWIG and Python. First, it is usually relatively easy to build a Python interface to an existing C library. With a little extra work, it is possible to improve the interface by adding a few support functions such as our Point extensions. Finally, once in Python, it is possible to encapsulate C libraries in new kinds of Python objects and classes. We built a simple Image base class and used it to plot mathematical functions and unstructured 2D mesh data---two entirely different tasks, yet easily accomplished with a small amount of Python code. If we later decided to use a different C library such as OpenGL, we could wrap it in SWIG, change the Image base class appropriately , and use the function and mesh plotting examples without modification. I think this is pretty cool.

Putting it all together

Finally, let's combine our heat equation solver and graphics module into a single application. To do this, we first need to know how to combine two different SWIG generated modules. When different SWIG modules need to be combined, there are a number of things you can do.

Merging modules

Two SWIG modules can be combined into a single module if you make an interface file like this :

```
%module package
%include pde.i
%include gd.i
```

This will combine everything in both interface files into a single super-module called "package". The advantage to this approach is that it is extremely quick and easy. The disadvantage is that the module names of "pde" and "gd" will be lost. If you had a bunch of scripts that relied on those names, they would no longer work. Thus, combining modules in this way is probably only a good idea if the modules are closely related.

Using dynamic loading

If your system supports dynamic loading, you can build each SWIG module into a separate dynamically loadable module and load each one individually into Python. This is the preferred approach if it is supported on your system. SWIG wrapper files declare virtually everything as "static" so using dynamic loading with multiple SWIG generated modules will not usually cause any namespace clashes.

Use static linking

As an alternative to dynamic loading, you can use a special form of the `%module` directive as follows :

```
%module package, pdec, gdc
%include embed.i
```

This will build a static version of Python with 3 C extension modules added (`package`, `pdec`, and `gdc`). When using this technique, the names of the modules refer to the low-level SWIG generated C/C++ modules. Since shadow classes are being used, these modules must have an extra 'c' appended to the name (thus, "pdec" and "gdc" instead of "pde" and "gd"). The extra modules specified with the `%modules` directive do not necessarily have to be SWIG-generated modules. In practice, almost any kind of Python module can be listed here. It should also be noted that extra modules names are completely ignored if the `embed.i` library file is not used.

Building large multi-module systems

By default, SWIG includes the C code for the SWIG type-checker and variable linking into every module. However, when, building systems involving large numbers of SWIG modules, common code such as the SWIG pointer type-checker and variable linking extensions can be shared if you run SWIG with the `-c` option. For example :

```
% swig -c -python graphics.i
% swig -c -python network.i
% swig -c -python analysis.i
% swig -c -python math.i
% gcc -c graphics_wrap.c network_wrap.c analysis_wrap.c math_wrap.c
% ld -shared graphics_wrap.o -lswigpy -o graphicsmodule.so
% ld -shared network_wrap.o -lswigpy -o networkmodule.so
% ld -shared analysis_wrap.o -lswigpy -o analysismodule.so
% ld -shared math_wrap.o -o -lswigpy mymathmodule.so
```

`swigpy` is a special purpose library that contains the SWIG pointer type checker and other support code (see the `Misc` subdirectory of the SWIG distribution). When used in this manner, the same support code will be used for all of the modules. The `swigpy` library can also be applied when static linking is being used. See the Advanced Topics chapter for more information about using SWIG with multiple modules.

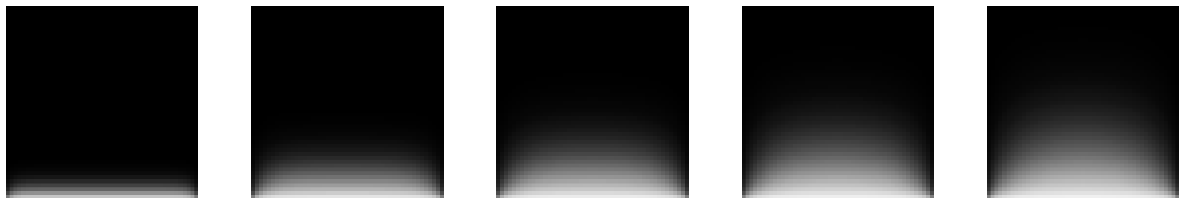
A complete application

The following Python script shows an application that combines our C++ heat equation solver, our gd library, and our Image base class that we developed.

```
# Solve the heat equation.
# Make a series of data files
# Make a movie of GIF images

from pde import *
from image import *
import string

# Image class
class HeatImg(Image):
       def __init__(self,h,width=300,height=300):
              Image.__init__(self,width,height,0.0,0.0,1.0,1.0)
              self.h = h
              # Create a greyscale colormap
              cmap = []
              for i in range(0,255):
                     cmap.append((i,i,i))
              self.colormap(cmap)
              self.cmin = 0.0
              self.cmax = 1.0
              self.imgno = 1
       def draw(self):
              self.clear(0)
              dx = 1.0/(self.h.grid.xpoints-2)
              dy = 1.0/(self.h.grid.ypoints-2)
              i = 1
              x = 0.0
              while i < self.h.grid.xpoints:
                     j = 1;
                     y = 0.0
                     while j < self.h.grid.ypoints:
                            c = int((self.h.grid[i][j]-self.cmin)/(self.cmax-
                                    self.cmin)*255)
                            self.solidbox(x,y+dy,x+dx,y,c)
                            j = j + 1
                            y = y + dy
                     i = i + 1
                     x = x + dx
```

```
                self.name = "image"+string.zfill(self.imgno,4)+".gif"
                self.imgno = self.imgno+1

# Set up an initial condition
def initcond(h):
        h.set_temp(0.0)
        nx = h.grid.xpoints
        for i in range(0,nx):
                h.grid[i][0] = 1.0

# Set up a problem and run it
h = Heat2d(50,50)

# Make an image object
img = HeatImg(h)

initcond(h)
fileno = 1

# Run it
for i in range(0,25):
        h.solve(100)
        h.dump("Dat"+str(fileno))
        img.show()
        print "time = ", h.time
        fileno = fileno+1

# Calculate average temperature and exit
sum = 0.0
for i in range(0,h.grid.xpoints):
        for j in range(0,h.grid.ypoints):
                sum = sum + h.grid[i][j]
avg = sum/(h.grid.xpoints*h.grid.ypoints)
print "Avg temperature = ",avg
```

When run, we now get a collection of datafiles and series of images like this :



Thus, we have a simple physics application that only takes about 1 page of Python code, runs a simulation, creates data files, and a movie of images. We can easily change any aspect of the simulation, interactively query variables and examine data. New procedures can be written and tested in Python and later implemented in C++ if needed. More importantly, we have an application that is actually fun to use and modify (well, at least I think so).

Exception handling

The SWIG `%except` directive can be used to create a user-definable exception handler in charge of converting exceptions in your C/C++ program 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:
    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 Python extension by specifying the following in an interface file :

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

When the C++ class throws a RangeError exception, our wrapper functions will catch it, turn it into a Python 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. Please see the chapter on exception handling for more details and using the `exception.i` library for writing language-independent exception handlers.

Python exceptions can be raised using the `PyErr_SetString()` function as shown above. The following table provides a list of the different Python exceptions available.

Built-in Python Exceptions

`PyExc_AttributeError`	Raised when an attribute reference or assignment fails. Usually raised when an invalid class attribute is accessed.
`PyExc_EOFError`	Indicates the end-of-file condition for I/O operations.
`PyExc_IOError`	Raised when an I/O occurs, e.g, 'file not found', 'permission denied', etc...
`PyExc_ImportError`	Raised when an 'import' statement fails.
`PyExc_IndexError`	Indicates a subscript out of range--usually for array and list indexing.
`PyExc_KeyError`	Raised when a dictionary key is not found in the set of existing keys. Could be used for hash tables and similar objects.
`PyExc_KeyboardInterrupt`	Raised when the user hits the interrupt key.
`PyExc_MemoryError`	Indicates a recoverable out of memory error.
`PyExc_NameError`	Raised to indicate a name not found.
`PyExc_OverflowError`	Raised when results of arithmetic computation is too large to be represented.
`PyExc_RuntimeError`	A generic exception for "everything else". Errors that don't fit into other categories.
`PyExc_SyntaxError`	Normally raised when the Python parser encounters a syntax error.
`PyExc_SystemError`	Used to indicate various system errors.
`PyExc_SystemExit`	A serious error, abandon all hope now.
`PyExc_TypeError`	Raised when an object is of invalid type.
`PyExc_ValueError`	Raised when an object has the right type, but an inappropriate value.
`PyExc_ZeroDivisionError`	Division by zero error.

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 Python's C API. The reader is advised to consult the Python reference manual or one of the books on Python. 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 overridden. A simple typemap might look like this :

```
%module example

%typemap(python,in) int {
       $target = (int) PyLong_AsLong($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 Python, "python" should be used as the language name. 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 `PyObject *` to the corresponding C datatype. Within the supporting C code, the variable `$source` contains the source data (the `PyObject` in this case) and `$target` contains the destination of a conversion.

When this example is compiled into a Python module, it will operate as follows :

```
>>> from example import *
>>> 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 Python typemaps here.

Python typemaps

The following typemap methods are available to Python modules :

`%typemap(python,in)`	Converts Python objects to input function arguments
`%typemap(python,out)`	Converts return value of a C function to a Python object
`%typemap(python,varin)`	Assigns a global variable from a Python object
`%typemap(python,varout)`	Returns a global variable as a Python object
`%typemap(python,freearg)`	Cleans up a function argument (if necessary)
`%typemap(python,argout)`	Output argument processing
`%typemap(python,ret)`	Cleanup of function return values
`%typemap(python,const)`	Creation of Python constants
`%typemap(memberin)`	Setting of C++ member data
`%typemap(memberout)`	Return of C++ member data
`%typemap(python,check)`	Checks function input values.

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)

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(python,in) char ** { ... }

// This typemap is applied only to char ** arguments named 'argv'
%typemap(python,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(python, 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(python,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 Python list to a char ***

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Python list object to be used as a `char **` object.

```
%module argv
```

```
// This tells SWIG to treat char ** as a special case
%typemap(python,in) char ** {
  /* Check if is a list */
  if (PyList_Check($source)) {
    int size = PyList_Size($source);
    int i = 0;
    $target = (char **) malloc((size+1)*sizeof(char *));
    for (i = 0; i < size; i++) {
      PyObject *o = PyList_GetItem($source,i);
      if (PyString_Check(o))
       $target[i] = PyString_AsString(PyList_GetItem($source,i));
      else {
       PyErr_SetString(PyExc_TypeError,"list must contain strings");
       free($target);
       return NULL;
      }
    }
    $target[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(python,freearg) char ** {
  free((char *) $source);
}

// This allows a C function to return a char ** as a Python list
%typemap(python,out) char ** {
  int len,i;
  len = 0;
  while ($source[len]) len++;
  $target = PyList_New(len);
  for (i = 0; i < len; i++) {
    PyList_SetItem($target,i,PyString_FromString($source[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];
}
%}
```

When this module is compiled, our wrapped C functions now operate 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
>>> get_args()
[‘Dave’, ‘Mike’, ‘Susan’, ‘John’, ‘Michelle’]
>>>
```

Our type-mapping makes the Python interface to these functions more natural and easy to use.

*Converting a Python file object to a FILE **

In our previous example involving gd-1.2, we had to write wrappers around `fopen()` and `fclose()` so that we could provide gd with a `FILE *` pointer. However, we could have used a typemap like this instead :

```
// Type mapping for grabbing a FILE * from Python

%typemap(python,in) FILE * {
  if (!PyFile_Check($source)) {
      PyErr_SetString(PyExc_TypeError, "Need a file!");
      return NULL;
  }
  $target = PyFile_AsFile($source);
}
```

Now, we can rewrite one of our earlier examples like this :

```
# Simple gd program

from gd import *

im = gdImageCreate(64,64)
black = gdImageColorAllocate(im,0,0,0)
white = gdImageColorAllocate(im,255,255,255)
gdImageLine(im,0,0,63,63,white)
f = open(“test.gif”,”w”)              # Create a Python file object
gdImageGif(im,f)                      # Pass to a C function as FILE *
f.close()
gdImageDestroy(im)
```

Using typemaps to return arguments

A common problem in some C programs is that values may be returned in arguments rather than in the return value of a function. For example :

```
/* Returns a status value and two values in out1 and out2 */
int spam(double a, double b, double *out1, double *out2) {
       ... Do a bunch of stuff ...
```

```
        *out1 = result1;
        *out2 = result2;
        return status;
};
```

A named 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(python,argout) double *OutValue {
        PyObject *o;
        o = PyFloat_FromDouble(*$source);
        if ((!$target) || ($target == Py_None)) {
                $target = o;
        } else {
                if (!PyList_Check($target)) {
                        PyObject *o2 = $target;
                        $target = PyList_New(0);
                        PyList_Append($target,o2);
                        Py_XDECREF(o2);
                }
                PyList_Append($target,o);
                Py_XDECREF(o);
        }
}
int spam(double a, double b, double *OutValue, double *OutValue);
```

With this typemap, we first check to see if any result exists. If so, we turn it into a list and append our new output value to it. If this is the only result, we simply return it normally. For our sample function, there are three output values so the function will return a list of 3 elements. As written, our function needs to take 4 arguments, the last two being pointers to doubles. We may not want to pass anything into these arguments if they are only used to hold output values so we could change this as follows :

```
%typemap(python,ignore) double *OutValue(double temp) {
        $target = &temp;       /* Assign the pointer to a local variable */
}
```

Now, in a Python script, we could do this :

```
>>> a = spam(4,5)
>>> print a
[0, 2.45, 5.0]
>>>
```

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(python,in) double[4](double temp[4]) {   // temp[4] becomes a local variable
  int i;
  if (PyTuple_Check($source)) {
    if (!PyArg_ParseTuple($source,”dddd”,temp,temp+1,temp+2,temp+3)) {
      PyErr_SetString(PyExc_TypeError,"tuple must have 4 elements");
      return NULL;
    }
    $target = &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 kind of scheme works fine.

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 the 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 Python 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 not be affected. However, you can write a typemap to match any sized array using the `ANY` keyword as follows :

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

During code generation, `$dim0` will be filled in with the real array dimension.

Useful Functions

When writing typemaps, it is often necessary to work directly with Python objects instead of using the conventional `PyArg_ParseTuple()` function that is usually used when writing Python extensions. However, there are a number of useful Python functions available for you to use.

Python Integer Conversion Functions

`PyObject *PyInt_FromLong(long l)`	Convert long to Python integer
`long PyInt_AsLong(PyObject *)`	Convert Python integer to long
`int PyInt_Check(PyObject *)`	Check if Python object is an integer

Python Floating Point Conversion Functions

`PyObject *PyFloat_FromDouble(double)`	Convert a double to a Python float
`double PyFloat_AsDouble(PyObject *)`	Convert Python float to a double
`int PyFloat_Check(PyObject *)`	Check if Python object is a float

Python String Conversion Functions

`PyObject *PyString_FromString(char *)`	Convert NULL terminated ASCII string to a Python string
`PyObject *PyString_FromStringAndSize(char *, int len)`	Convert a string and length into a Python string. May contain NULL bytes.
`int PyString_Size(PyObject *)`	Return length of a Python string
`char *PyString_AsString(PyObject *)`	Return Python string as a NULL terminated ASCII string.
`int PyString_Check(PyObject *)`	Check if Python object is a string

Python List Conversion Functions

`PyObject *PyList_New(int size)`	Create a new list object
`int PyList_Size(PyObject *list)`	Get size of a list
`PyObject *PyList_GetItem(PyObject *list, int i)`	Get item i from list
`int PyList_SetItem(PyObject *list, int i, PyObject *item)`	Set list[i] to item.
`int PyList_Insert(PyObject *list, int i, PyObject *item)`	Inserts item at list[i].

Python List Conversion Functions

`int PyList_Append(PyObject *list, PyObject *item)`	Appends item to list
`PyObject *PyList_GetSlice(PyObject *list, int i, int j)`	Returns list[i:j]
`int PyList_SetSlice(PyObject *list, int i, int j, PyObject *list2)`	Sets list[i:j] = list2
`int PyList_Sort(PyObject *list)`	Sorts a list
`int PyList_Reverse(PyObject *list)`	Reverses a list
`PyObject *PyList_AsTuple(PyObject *list)`	Converts a list to a tuple
`int PyList_Check(PyObject *)`	Checks if an object is a list

Python Tuple Functions

`PyObject *PyTuple_New(int size)`	Create a new tuple
`int PyTuple_Size(PyObject *t)`	Get size of a tuple
`PyObject *PyTuple_GetItem(PyObject *t,int i)`	Get object t[i]
`int PyTuple_SetItem(PyObject *t, int i, PyObject *item)`	Set t[i] = item
`PyObject *PyTuple_GetSlice(PyObject *t,int i int j)`	Get slice t[i:j]
`int PyTuple_Check(PyObject *)`	Check if an object is a tuple

Python File Conversions

`PyObject *PyFile_FromFile(FILE *f)`	Convert a FILE * to a Python file object
`FILE *PyFile_AsFile(PyObject *)`	Return FILE * from a Python object
`int PyFile_Check(PyObject *)`	Check if an object is a file

Standard typemaps

The following typemaps show how to convert a few common kinds of objects between Python and C (and to give a better idea of how typemaps work)

Function argument typemaps

int,
short,
long
	`%typemap(python,in) int,short,long {`
	`  if (!PyInt_Check($source)) {`
	`    PyErr_SetString(PyExc_TypeError,"not an integer”);`
	`    return NULL;`
	`  }`
	`  $target = ($type) PyInt_AsLong($source);`
	`}`

float,
double
	`%typemap(python,in) float,double {`
	`  if (!PyFloat_Check($source)) {`
	`    PyErr_SetString(PyExc_TypeError,”not a float”);`
	`    return NULL;`
	`  }`
	`  $target = ($type) PyFloat_AsDouble($source);`
	`}`

char *	`%typemap(python,in) char * {`
	`  if (!PyString_Check($source)) {`
	`    PyErr_SetString(PyExc_TypeError,”not a string”);`
	`    return NULL;`
	`  }`
	`  $target = PyString_AsString($source);`
	`}`

FILE *	`%typemap(python,in) FILE * {`
	`  if (!PyFile_Check($source)) {`
	`    PyErr_SetString(PyExc_TypeError,”not a file”);`
	`    return NULL;`
	`  }`
	`  $target = PyFile_AsFile($source);`
	`}`

Function return typemaps

int,
short
	`%typemap(python,out) int,short {`
	`    $target = PyBuild_Value(“i”, ($type) $source);`
	`}`

long	`%typemap(python,out) long {`
	`    $target = PyBuild_Value(“l”,$source);`
	`}`

float,
double
	`%typemap(python,out) float,double {`
	`    $target = PyBuild_Value(“d”,($type) $source);`
	`}`

Function return typemaps

`char *`	`%typemap(python,out) char * {`
	`    $target = PyBuild_Value("s",$source);`
	`}`

`FILE *`	`%typemap(python,out) FILE * {`
	`    $target = PyFile_FromFile($source);`
	`}`

Pointer handling

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 .

SWIG Pointer Conversion Functions

`void SWIG_MakePtr(char *str, void *ptr, char *type)`	Makes a pointer string and saves it in `str`, which must be large enough to hold the result. `ptr` contains the pointer value and `type` is the string representation of the type.
`char *SWIG_GetPtr(char *str, void **ptr, char *type)`	Attempts to read a pointer from the string `str`. `ptr` is the address of the pointer to be created and `type` is the expected type. If `type` is NULL, then any pointer value will be accepted. On success, this function returns NULL. On failure, it returns the pointer to the invalid portion of the pointer string.

These functions can be used in typemaps. For example, the following typemap makes an argument of "`char *buffer`" accept a pointer instead of a NULL-terminated ASCII string.

```
%typemap(python,in) char *buffer {
        PyObject *o;
        char     *str;
        if (!PyString_Check(o)) {
                PyErr_SetString(PyExc_TypeError,"not a string");
                return NULL;
        }
        str = PyString_AsString(o);
        if (SWIG_GetPtr(str, (void **) &$target, "$mangle")) {
                PyErr_SetString(PyExc_TypeError,"not a pointer");
                return NULL;
        }
}
```

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 Typemaps chapter for more information about using library files.

Implementing C callback functions in Python

Now that you're an expert, we will implement simple C callback functions in Python and use them in a C++ code.

Let's say that we wanted to write a simple C++ 2D plotting widget layered on top of the gd-1.2 library. A class definition might look like this :

```
// -------------------------------------------------------------------
// Create a C++ plotting "widget" using the gd-1.2 library by Thomas Boutell
//
// This example primarily illustrates how callback functions can be
// implemented in Python.
// -------------------------------------------------------------------

#include <stdio.h>
extern "C" {
#include "gd.h"
}

typedef double (*PLOTFUNC)(double, void *);

class PlotWidget {
private:
  double      xmin,ymin,xmax,ymax;         // Plotting range
  PLOTFUNC    callback;                    // Callback function
  void       *clientdata;                  // Client data for callback
  int         npoints;                     // Number of points to plot
  int         width;                       // Image width
  int         height;                      // Image height
  int         black,white;                 // Some colors
  gdImagePtr  im;                          // Image pointer
  void        transform(double,double,int&,int&);
public:
  PlotWidget(int w, int h,double,double,double,double);
  ~PlotWidget();
  void set_method(PLOTFUNC func, void *clientdata);    // Set callback method
  void set_range(double,double,double,double);         // Set plot range
  void set_points(int np) {npoints = np;}              // Set number of points
  void plot();                                         // Make a plot
  void save(FILE *f);                                  // Save a plot to disk
};
```

The widget class hides all of the underlying implementation details so this could have just as easily been implemented on top of OpenGL, X11 or some other kind of library. When used in C++, the widget works like this :

```
// Simple main program to test out our widget
#include <stdio.h>
#include "widget.h"
#include <math.h>
```

```
// Callback function
double my_func(double a, void *clientdata) {
  return sin(a);
}

int main(int argc, char **argv) {
  PlotWidget *w;
  FILE *f;

  w = new PlotWidget(500,500,-6.3,-1.5,6.3,1.5);
  w->set_method(my_func,0);             // Set callback function
  w->plot();                            // Make plot
  f = fopen("plot.gif","w");
  w->save(f);
  fclose(f);
  printf("wrote plot.gif\n");
}
```

Now suppose that we wanted to use our widget interactively from Python. While possible, it is going to be difficult because we would really like to implement the callback function in Python, not C++. We also don't want to go in and hack or C++ code to support this. Fortunately, you can do it with SWIG using the following interface file :

```
// SWIG interface to our PlotWidget
%module plotwidget
%{
#include "widget.h"
%}

// Grab a Python function object as a Python object.
%typemap(python,in) PyObject *pyfunc {
  if (!PyCallable_Check($source)) {
      PyErr_SetString(PyExc_TypeError, "Need a callable object!");
      return NULL;
  }
  $target = $source;
}

// Type mapping for grabbing a FILE * from Python
%typemap(python,in) FILE * {
  if (!PyFile_Check($source)) {
      PyErr_SetString(PyExc_TypeError, "Need a file!");
      return NULL;
  }
  $target = PyFile_AsFile($source);
}

// Grab the class definition
%include widget.h

%{
/* This function matches the prototype of the normal C callback
   function for our widget. However, we use the clientdata pointer
   for holding a reference to a Python callable object. */

static double PythonCallBack(double a, void *clientdata)
{
   PyObject *func, *arglist;
   PyObject *result;
   double    dres = 0;

   func = (PyObject *) clientdata;               // Get Python function
   arglist = Py_BuildValue("(d)",a);             // Build argument list
   result = PyEval_CallObject(func,arglist);     // Call Python
   Py_DECREF(arglist);                           // Trash arglist
   if (result) {                                 // If no errors, return double
     dres = PyFloat_AsDouble(result);
   }
   Py_XDECREF(result);
   return dres;
}
%}

// Attach a new method to our plot widget for adding Python functions
%addmethods PlotWidget {
   // Set a Python function object as a callback function
   // Note : PyObject *pyfunc is remapped with a typempap
   void set_pymethod(PyObject *pyfunc) {
     self->set_method(PythonCallBack, (void *) pyfunc);
     Py_INCREF(pyfunc);
   }
}
```

While this is certainly not a trivial SWIG interface file, the results are quite cool. Let’s try out our new Python module :

```
# Now use our plotting widget in variety of ways

from plotwidget import *
from math import *

# Make a plot using a normal Python function as a callback
def func1(x):
       return 0.5*sin(x)+0.25*sin(2*x)+0.125*cos(4*x)

print "Making plot1.gif..."
# Make a widget and set callback
w = PlotWidget(500,500,-10,-2,10,2)
w.set_pymethod(func1)                     # Register our Python function
w.plot()
f = open("plot1.gif","w")
w.save(f)
f.close()

# Make a plot using an anonymous function

print "Making plot2.gif..."
w1 = PlotWidget(500,500,-4,-1,4,16)
w1.set_pymethod(lambda x: x*x)            # Register x^2 as a callback
w1.plot()
f = open("plot2.gif","w")
w1.save(f)
f.close()

# Make another plot using a built-in function
```

```
print "Making plot3.gif..."
w2 = PlotWidget(500,500,-7,-1.5,7,1.5)
w2.set_pymethod(sin)                      # Register sin(x) as a callback
w2.plot()
f = open("plot3.gif","w")
w2.save(f)
f.close()
```

The “plot” method for each widget is written entirely in C++ and assumes that it is calling a callback function written in C/C++. Little does it know that we have actually implemented this function in Python. With a little more work, we can even write a simple function plotting tool :

```
# Plot a function and spawn xv

import posix
import sys
import string
from plotwidget import *
from math import *

line = raw_input("Enter a function of x : ")
ranges = string.split(raw_input("Enter xmin,ymin,xmax,ymax :"),",")

print "Making a plot..."
w = PlotWidget(500,500,string.atof(ranges[0]),string.atof(ranges[1]),
               string.atof(ranges[2]),string.atof(ranges[3]))

# Turn user input into a Python function
code = "def func(x): return " + line
exec(code)

w.set_pymethod(func)
w.plot()
f = open("plot.gif","w")
w.save(f)
f.close()
posix.system("xv plot.gif &")
```

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);
```

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 :

```
>>> gptr = create_grid2d()         # Returns a Grid2d from somewhere
>>> g = Grid2dPtr(gptr)            # 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)
>>> 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):
                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 alot of problems, but it hurts quite alot if you accidently 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 alot 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.

SWIG and Tcl 10

This chapter discusses SWIG's support of Tcl. SWIG supports Tcl versions 7.3 and newer, including Tcl 8.0. Tk 3.6 and newer can also be used. However, for the best results you should consider using Tcl 7.5/Tk4.1 or later.

Preliminaries

You will need to install Tcl/Tk on your system if you haven't done so already. If you are using Tcl 7.5 or newer, you should also determine if your system supports dynamic loading and shared libraries. SWIG will work with or without this, but the compilation process varies.

Running SWIG

To build a Tcl module, run swig using the `-tcl` option as follows :

```
swig -tcl example.i
```

This will produce 2 files. The first file, `example_wrap.c,` contains all of the C code needed to build your Tcl module. The second file contains supporting documentation and may be named `example_wrap.doc`, `example_wrap.html`, `example_wrap.tex`, etc... To build a Tcl extension you will need to compile the `example_wrap.c` file and link it with the rest of your program (and possibly Tcl itself).

Additional SWIG options

The following options are also available with the Tcl module :

```
-tcl8         Produce Tcl8.0 native wrappers (use in place of -tcl).
-module       Set the module name.
-namespace    Use [incr Tcl] namespaces.
-prefix pkg   Set a package prefix of 'pkg'. This prefix will be
              attached to each function.
-htcl tcl.h   Set name of Tcl header file.
-htk tk.h     Set name of Tk header file.
-plugin       Generate additional code for the netscape plugin.
-noobject     Omit object oriented extensions (compatibility with SWIG 1.0)
```

Many of these options will be described later.

Getting the right header files and libraries

In order to compile Tcl/Tk extensions, you will need to locate the "`tcl.h`" and "`tk.h`" header files. These are usually located in `/usr/local/include`. You will also need to locate the Tcl/

Tk libraries `libtcl.a` and `libtk.a`. These are usually located in `/usr/local/lib`.
When locating the right header and libraries files, double check to make sure the files are the correct version and form a matching pair. SWIG works with the following Tcl/Tk releases.

```
Tcl 7.3, Tk 3.6
Tcl 7.4, Tk 4.0
Tcl 7.5, Tk 4.1
Tcl 7.6, Tk 4.2
Tcl 8.0a2, Tk 8.0a2
```

Do not mix versions. Although the code might compile if you do, it will usually core dump mysteriously. By default, SWIG looks for the header files "`tcl.h`" and "`tk.h`", but your installed version of Tcl/Tk may use slightly different names such as "`tcl7.5.h`" and "`tk4.1.h`". If you need to use different header files, you can use the `-htcl` and `-htk` options as in :

```
swig -tcl -htcl tcl7.5.h -htk tk4.1.h example.i
```

If you are installing Tcl/Tk yourself, it is often easier to set up a symbolic links between `tcl.h` and the header files for the latest installed version. You might also be able to make symbolic links to the correct files in your working directory.

Compiling a dynamic module (Unix)

To compile a dynamically loadable module, you will need to compile your SWIG extension into a shared library. This usually looks something like the following (shown for Linux).

```
unix > swig -tcl example.i
unix > gcc -fpic example_wrap.c example.c -I/usr/local/include
unix > gcc -shared example.o example_wrap.o -o example.so       # Linux
```

Unfortunately, the process of building of building shared libraries varies on every single machine. SWIG will try to guess when you run configure, but it isn't always successful. It's always a good idea to read the man pages on the compiler/linker to find out more information.

Using a dynamic module

To use a dynamic module, you will need to load it using the Tcl load command as follows :

```
load ./example.so example
```

The first argument is the name of the shared library while the second argument is the name of the module (the same as what you specified with the `%module` directive). As alternative, you can turn your module into a Tcl package. See the section on configuration management at the end of this chapter for details.

Static linking

If your machine does not support dynamic loading or you've tried to use it without success, you can build new versions of `tclsh` (the Tcl shell) or `wish` (Tcl/Tk shell) with your extensions added. To do this, use SWIG's `%include` directive as follows :

```
%module mymodule
... declarations ...
```

```
%include tclsh.i     // Support code for rebuilding tclsh
```

To rebuild tclsh, you will need to compile as follows :

```
unix > swig -tcl example.i
unix > gcc example_wrap.c example.c -I/usr/local/include -L/usr/local/lib -ltcl -ldl \
       -lm -o my_tclsh
```

Alternatively, you can use SWIG's `-l` option to add the tclsh.i library file without modifying the interface file. For example :

```
unix > swig -tcl -ltclsh.i example.i
unix > gcc example_wrap.c example.c -I/usr/local/include -L/usr/local/lib -ltcl -ldl \
       -lm -o my_tclsh
```

The `-ldl` option will be required if your Tcl/Tk supports dynamic loading. On some machines (most notably Solaris),it will also be necessary to add `-lsocket -lnsl` to the compile line. This will produce a new version of `tclsh` that is identical to the old one, but with your extensions added.

If you are using Tk, you will want to rebuild the `wish` executable instead. This can be done as follows :

```
%module mymodule
... declarations ...

%include wish.i      // Support code for rebuilding wish
```

The compilation process is similar as before, but now looks like this :

```
unix > swig -tcl example.i
unix > gcc example_wrap.c example.c -I/usr/local/include -L/usr/local/lib -ltk -ltcl \
       -lX11 -ldl -lm -o my_wish
```

In this case you will end up with a new version of the `wish` executable with your extensions added. Make sure you include `-ltk`, `-ltcl`, and `-lX11` in the order shown.

Compilation problems

Tcl is one of the easiest languages to compile extensions for. The Tcl header files should work without problems under C and C++. Perhaps the only tricky task is that of compiling dynamically loadable modules for C++. If your C++ code has static constructors, it is unlikely to work at all. In this case, you will need to build new versions of the `tclsh` or `wish` executables instead. You may also need to link against the `libgcc.a`, `libg++.a`, and `libstdc++.a` libraries (assuming g++).

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 [incr Tcl] namespaces

Alternatively, you can have SWIG install your module into an [incr 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, the resulting wrapper code can be compiled under both Tcl and [incr Tcl]. When compiling under Tcl, the namespace will turn into a package prefix such as in `Foo_bar`. When running under [incr Tcl], it will be something like `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++ 4.x 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.

Basic Tcl Interface

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, things are a little more complicated. For the following C datatypes, SWIG will use Tcl's variable linking mechanism to provide direct access :

```
int, unsigned int,
double,
char *,
```

When used in an interface file and script, it will operate as follows :

```
// 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
```

For all other C datatypes, SWIG will generate a pair of set/get functions. For example :

```
// example.i
short My_short;
```

will be accessed in Tcl as follows :

```
puts [My_short_get]                # Get value of global variable
My_short_set 5.5                   # Set value of global variable
```

While not the most elegant solution, this is the only solution for now. Tcl's normal variable linking mechanism operates directly on a variables and would not work correctly on datatypes other than the 3 basic datatypes supported by Tcl (`int`, `double`, and `char *`).

Constants

Constants are created as read-only variables. For odd datatypes (not supported by the variable linking mechanism), SWIG generates a string representation of the constant and use it instead (you shouldn't notice this however since everything is already a string in Tcl). It is never necessary to use a special "get" function with constants. Unlike Tcl variables, constants can contain pointers, structure addresses, function pointers, etc...

Pointers

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

```
_100f8e2_Vector_p
```

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

```
_0_Vector_p
```

In Tcl 8.0, pointers are represented using a new type of Tcl object, but the string representation is the same (and is interchangable). As a general, direct manipulation of pointer values is discouraged.

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)
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_List_p
%
```

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 1.1 also generates an object oriented interface that can be used in addition to the basic interface just described here.

The object oriented interface

With SWIG 1.1, a new object oriented interface to C structures and C++ classes is supported. This interface supplements the low-level SWIG interface already defined--in fact, both can be used simultaneously. To illustrate this interface, consider our 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 ‘o’

MyObject o -this $objptr  # Turn a pointer to an existing C++ object into a
                          # Tcl object ‘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, we can create new List objects as follows :

```
List l                    # Create a new list l

set listptr [new_List]    # Create a new List using low level interface
List l2 -this $listptr    # Turn it into a List object named ‘l2’

set l3 [List]             # Create a new list. The name of the list is in $l3

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

Now 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_List_p
```

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 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... (this is the pointer model)

```
% 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’
```

SWIG will automatically call the corresponding C/C++ destructor, with one caveat--SWIG will not destroy an object if you created it from an already existing pointer (if you called the constructor using the -this option). Since the pointer already existed when you created the Tcl object, Tcl doesn’t own the object so it would probably be a bad idea to destroy it.

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.

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).
```

Performance concerns and disabling the object oriented interface

The object oriented interface is mainly provided for ease of programming at the expense of introducing more overhead and increased code size (C code that is). If you are concerned about these issues use the basic SWIG interface instead. It provides direct access and is much faster. As it turns out, it is possible to build an object oriented interface directly in Tcl as well--an example we'll return to a little later.

To disable the object oriented interface, run SWIG with the `-noobject` option. This will strip out all of the extra code and produce only the low-level interface.

About the examples

The next few sections cover Tcl examples of varying complexity. These are primarily designed to illustrate how SWIG can be used to integrate C/C++ and Tcl in a variety of ways. Some of the things that will be covered are :

- Controlling C programs with Tcl
- Building C data structures in Tcl.
- Use of C objects with Tk
- Wrapping a C library (OpenGL in this case)
- Accessing arrays and other common data structures
- Using Tcl to build new Tcl interfaces to C programs.
- Modifying SWIG's handling of datatypes.
- And a bunch of other cool stuff.

Binary trees in Tcl

In this example, we show Tcl can be used to manipulate binary trees implemented in C. This will involve accessing C data structures and functions.

C files

We will build trees using the following C header file :

```
/* tree.h */
typedef struct Node Node;
struct Node {
       char          *key;
       char          *value;
       Node          *left;
       Node          *right;
};

typedef struct Tree {
       Node          *head;        /*  Starting node */
       Node          *z;           /* Ending node (at leaves) */
} Tree;
```

```
extern Node *new_Node(char *key, char *value);
extern Tree *new_Tree();
```

The C functions to create new nodes and trees are as follows :

```
/* File : tree.c */
#include <string.h>
#include “tree.h”
Node *new_Node(char *key, char *value) {
       Node *n;
       n = (Node *) malloc(sizeof(Node));
       n->key = (char *) malloc(strlen(key)+1);
       n->value = (char *) malloc(strlen(value)+1);
       strcpy(n->key,key);
       strcpy(n->value,value);
       n->left = 0;
       n->right = 0;
       return n;
};
Tree *new_Tree() {
       Tree *t;
       t = (Tree *) malloc(sizeof(Tree));
       t->head = new_Node(““,”__head__”);
       t->z = new_Node(“__end__”,”__end__”);
       t->head->right = t->z;
       t->z->left = t->z;
       t->z->right = t->z;
       return t;
}
```

Making a quick a dirty Tcl module

To make a quick Tcl module with these functions, we can do the following :

```
// file : tree.i
%module tree
%{
#include “tree.h”
%}
%include tree.h              // Just grab header file since it’s fairly simple
```

To build the module, run SWIG as follows and compile the resulting output :

```
% swig -tcl -ltclsh.i tree.i
% gcc tree.c tree_wrap.c -I/usr/local/include -L/usr/local/lib -ltcl -lm -o my_tclsh
```

We can now try out the module interactively by just running the new ‘`my_tclsh`’ executable.

```
unix > my_tclsh
% set t [new_Tree]                  # Create a new tree
_8053198_Tree_p
% set n [Tree_head_get $t]          # Get first node
_80531a8_Node_p
% puts [Node_value_get $n]          # Get its value
__head__
% Node -this $n
% $n cget -value                    # Altenative method for getting value
__head__
```

Building a C data structure in Tcl

Given our simple Tcl interface, it is easy to write Tcl functions for building up a C binary tree. For example :

```
# Insert an item into a tree
proc insert_tree {tree key value} {
    set tree_head [Tree_head_get $tree]
    set tree_z [Tree_z_get $tree]
    set p $tree_head
    set x [Node_right_get $tree_head]
    while {[Node_key_get $x] != “__end__”} {
        set p $x
        if {$key < [Node_key_get $x]} {
              set x [Node_left_get $x]
        } {
              set x [Node_right_get $x]
        }
    }
    set x [new_Node $key $value]
    if {$key < [Node_key_get $p]} {
        Node_left_set $p $x
    } {
        Node_right_set $p $x
    }
    Node_left_set $x $tree_z
    Node_right_set $x $tree_z
}

# Search tree and return all matches
proc search_tree {tree key} {
    set tree_head [Tree_head_get $tree]
    set tree_z [Tree_z_get $tree]
    set found {}
    set x [Node_right_get $tree_head]
    while {[Node_key_get $x] != "__end__"} {
        if {[Node_key_get $x] == $key} {
            lappend found [Node_value_get $x]
        }
        if {$key < [Node_key_get $x]} {
            set x [Node_left_get $x]
        } {
            set x [Node_right_get $x]
        }
    }
    return $found
}
```

While written in Tcl, these functions are building up a real C binary tree data structure that could be passed into other C function. For example, we could write a function that globs an entire directory and builds a tree structure as follows :

```
# Insert all of the files in pathname into a binary tree
```

```
proc build_dir_tree {tree pathname} {
    set error [catch {set filelist [glob -nocomplain $pathname/*]}]
    if {$error == 0} {
        foreach f $filelist {
            if {[file isdirectory $f] == 1} {
                insert_tree $tree [file tail $f] $f
                if {[file type $f] != "link"} {build_dir_tree $tree $f}
            } {
                insert_tree $tree [file tail $f] $f
            }
        }
    }
}
```

We can test out our function interactively as follows :

```
% source tree.tcl
% set t [new_Tree]          # Create a new tree
_80533c8_Tree_p
% build_dir_tree $t /home/beazley/SWIG/SWIG1.1
% search_tree $t tcl
/home/beazley/SWIG/SWIG1.1/Examples/tcl /home/beazley/SWIG/SWIG1.1/swig_lib/tcl
%
```

Implementing methods in C

While our Tcl methods may be fine for small problems, it may be faster to reimplement the insert and search methods in C :

```
void insert_tree(Tree *t, char *key, char *value) {
       Node *p;
       Node *x;

       p = t->head;
       x = t->head->right;
       while (x != t->z) {
              p = x;
              if (strcmp(key,x->key) < 0)
                     x = x->left;
              else
                     x = x->right;
       }
       x = new_Node(key,value);
       if (strcmp(key,p->key) < 0)
              p->left = x;
       else
              p->right = x;
       x->left = t->z;
       x->right = t->z;
}
```

To use this function in Tcl, simply put a declaration into the file `tree.h` or `tree.i`.

When reimplemented in C, the underlying Tcl script may not notice the difference. For example, our directory subroutine would not care if `insert_tree` had been written in Tcl or C. Of course, by writing this function C, we will get significantly better performance.

Building an object oriented C interface

So far our tree example has been using the basic SWIG interface. With a little work in the interface file, we can improve the interface a little bit.

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

%include tree.h
%{

  /* Function to count up Nodes */
  static  int count_nodes(Node *n, Node *end) {
    if (n == end) return 0;
    return 1+count_nodes(n->left,end)+count_nodes(n->right,end);
  }

%}

// Attach some new methods to the Tree structure

%addmethods Tree {
   void insert(char *key, char *value) {
        insert_tree(self,key,value);
   }
   char *search(char *key) {
        return search_tree(self,key);
   }
   char *findnext(char *key) {
     return find_next(self,key);
   }
   int count() {
     return count_nodes(self->head->right,self->z);
   }
   Tree();        // This is just another name for new_Tree
}
```

The `%addmethods` directive can be used to attach methods to existing structures and classes. In this case, we are attaching some methods to the `Tree` structure. Each of the methods are simply various C functions we have written for accessing trees. This type of interface file comes in particularly handy when using the Tcl object oriented interface. For example, we can rewrite our directory globber as follows :

```
proc build_dir_tree {tree pathname} {
    set error [catch {set filelist [glob -nocomplain $pathname/*]}]
    if {$error == 0} {
        foreach f $filelist {
            if {[file isdirectory $f] == 1} {
                $tree insert [file tail $f] $f      # Note new calling method
                if {[file type $f] != "link"} {build_dir_tree $tree $f}
            } {
                $tree insert [file tail $f] $f
            }
        }
    }
}
```

Now using it :

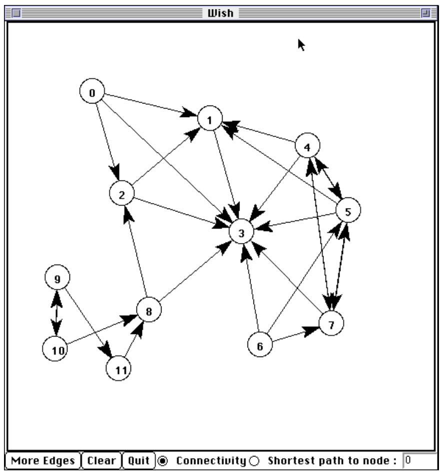
```
% source tree.tcl
% Tree t                    # Create a new Tree object
_8054610_Tree_p
% build_dir_tree t /home/beazley/SWIG/SWIG1.1
% t count
1770
% t search glaux.i
/home/beazley/SWIG/SWIG1.1/Examples/OpenGL/glaux.i
% t search typemaps
/home/beazley/SWIG/SWIG1.1/Examples/perl5/typemaps
% t findnext typemaps
/home/beazley/SWIG/SWIG1.1/Examples/python/typemaps
% t findnext typemaps
/home/beazley/SWIG/SWIG1.1/Examples/tcl/typemaps
% t findnext typemaps
None
%
```

With a little extra work, we've managed to turn an ordinary C structure into a class-like object in Tcl.

Building C/C++ data structures with Tk

Given the direct access to C/C++ objects provided by SWIG, it can be possible to use Tk to interactively build a variety of interesting data structures. To do this, it is usually useful to maintain a mapping between canvas items and an underlying C data structure. This is done using associative arrays to map C pointers to canvas items and canvas items back to pointers.

Suppose that we have a C program for manipulating directed graphs and that we wanted to provide a Tk interface for building graphs using a ball-and-stick model such as the following :



The SWIG interface file for this might look something like this :

```
%module graph
%{
```

```
#include "graph.h"
%}

/* Get a node's number */
int      GetNum(Node *n);                        /* Get a node's number         */
AdjList *GetAdj(Node *n);;                       /* Get a node's adjacency list */
AdjList *GetNext(AdjList *l);                    /* Get next node in adj. list  */
Node    *GetNode(AdjList *l);                    /* Get node from adj. list     */
Node    *new_node();                             /* Make a new node             */
void     AddLink(Node *v1, Node *v2);            /* Add an edge                 */
... etc ...
```

The interface file manipulates `Node` and `AdjList` structures. The precise implementation of these doesn't matter here--in fact SWIG doesn't even need it. Within a Tcl/Tk script however, we can keep track of these objects as follows :

```
# Make a new node and put it on the canvas
proc mkNode {x y} {
    global nodeX nodeY nodeP nodeMap nodeList edgeFirst edgeSecond
    set new [.c create oval [expr $x-15] [expr $y-15] \
               [expr $x+15] [expr $y+15] -outline black \
               -fill white -tags node]
    set newnode [new_node]                        ;# Make a new C Node
    set nnum [GetNum $newnode]                    ;# Get our node's number
    set id [.c create text [expr $x-4] [expr $y+10] \
            -text $nnum -anchor sw -tags nodeid]
    set nodeX($new) $x                            ;# Save coords of canvas item
    set nodeY($new) $y
    set nodeP($new) $newnode                      ;# Save C pointer
    set nodeMap($newnode) $new                    ;# Map pointer to Tk widget
    set edgeFirst($new) {}
    set edgeSecond($new) {}
    lappend nodeList $new                         ;# Keep a list of all C
                                                  ;# Pointers we've made
}

# Make a new edge between two nodes and put an arrow on the canvas
proc mkEdge {first second new} {
    global nodeP edgeFirst edgeSecond
    set edge [mkArrow $first $second]             ;# Make an arrow
    lappend edgeFirst($first) $edge               ;# Save information
    lappend edgeSecond($second) $edge
    if {$new == 1} {
# Now add an edge within our C data structure
       AddLink $nodeP($first) $nodeP($second)     ;# Add link to C structure
    }
}
```

In these functions, the array `nodeP()` allows us to associate a particular canvas item with a C object. When manipulating the graph, this makes it easy to move from the Tcl world to C. A second array, `nodeMap()` allows us to go the other way--mapping C pointers to canvas items. A list `nodeList` keeps track of all of the nodes we've created just in case we want to examine all of the nodes. For example, suppose a C function added more edges to the graph. To reflect the new state of the graph, we would want to add more edges to the Tk canvas. This might be accomplished as follows :

```
# Look at the C data structure and update the canvas
proc mkEdges {} {
    global nodeX nodeY nodeP nodeMap nodeList edgeFirst edgeSecond
    unset edgeFirst
    unset edgeSecond
    .c delete arc              # Edges have been tagged with arc (not shown)

    foreach node $nodeList {   ;# clear old lists
       set edgeFirst($node) {}
       set edgeSecond($node) {}
    }
    foreach node $nodeList {                    ;# Go through all of the nodes
       set v $nodeP($node)                      ;# Get the node pointer
       set v1 [GetAdj $v]                       ;# Get its adjacency list
       while {$v1 != "NULL"} {
           set v2 [GetNode $v1]                 ;# Get node pointer
           set w $nodeMap($v2)                  ;# Get canvas item
           mkEdge $node $w 0                    ;# Make an edge between them
           set v1 [GetNext $v1]                 ;# Get next node
       }
    }
}
```

This function merely scans through all of the nodes and walks down the adjacency list of each one. The `nodeMap()` array maps C pointers onto the corresponding canvas items. We use this to construct edges on the canvas using the `mkEdge` function.

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.

Building a simple OpenGL module

In this example, we consider building a SWIG module out of the OpenGL graphics library. The OpenGL library consists of several hundred functions for building complex 3D images. By wrapping this library, we will be able to play with it interactively from a Tcl interpreter.

Required files

To build an OpenGL module, you will need to have some variant of OpenGL installed on your machine. If unavailable, the Mesa graphics library is an OpenGL clone that runs on most machines that support X11. We will use the "GL/gl.h", "GL/glu.h", and the GL Auxilliary libraries.

Wrapping gl.h

The first step is to build an interface from the gl.h file. To do this, follow these steps :

- Copy the file `gl.h` to a file `gl.i` which will become the interface.
- Edit the `gl.i` file by taking out unneeded C preprocessor directives and any other clutter that you find.
- Put the following code at the beginning of the `gl.i` file

```
// gl.i : SWIG file for OpenGL
%module gl
%{
#include <GL/gl.h>
%}

... Rest of edited gl.h file here ...
```

A critical part of this first step is making sure you have the proper set of typedefs in the `gl.i` file. The first part of the file should include definitions such as the following :

```
typedef unsigned int GLenum;
typedef unsigned char GLboolean;
typedef unsigned int GLbitfield;
typedef signed char GLbyte;
typedef short GLshort;
typedef int GLint;
typedef int GLsizei;
typedef unsigned char GLubyte;
typedef unsigned short GLushort;
typedef unsigned int GLuint;
typedef float GLfloat;
typedef float GLclampf;
typedef double GLdouble;
typedef double GLclampd;
typedef void GLvoid;
```

Wrapping glu.h

Next we write a module for the glu library. The procedure is essentially identical to `gl.h`--that is, we'll copy the header file and edit it slightly. In this case, `glu.h` contains a few functions involving function pointers and arrays. These may generate errors or warnings. As a result, we can simply edit those declarations out. Fortunately, there aren't many declarations like this. If access is required later, the problem can often be fixed with typemaps and helper functions. The final `glu.i` file will look something like this :

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

... rest of edited glu.h file here ...
```

Given these two files, we have a fairly complete OpenGL package. Unfortunately, we still don't have any mechanism for opening a GL window and creating images. To to this, we can wrap the OpenGL auxilliary library which contains functions to open windows and draw various kinds of objects (while not the most powerful approach, it is the simplest to implement and works on most machines).

Wrapping the aux library

Wrapping the aux library follows exactly the same procedure as before. You will create a file `glaux.i` such as the following :

```
// File :glaux.i
%module glaux
%{
#include "glaux.h"
%}

... Rest of edited glaux.h file ...
```

A few helper functions

Finally, to make our library a little easier to use, we need to have a few functions to handle arrays since quite a few OpenGL calls use them as arguments. Small 4 element arrays are particularly useful so we'll create a few helper functions in a file called `help.i.`

```
// help.i : OpenGL helper functions

%inline %{
GLfloat *newfv4(GLfloat a,GLfloat b,GLfloat c,GLfloat d) {
  GLfloat *f;
  f = (GLfloat *) malloc(4*sizeof(GLfloat));
  f[0] = a; f[1] = b; f[2] = c; f[3] = d;
  return f;
}
void setfv4(GLfloat *fv, GLfloat a, GLfloat b, GLfloat c, GLfloat d) {
  fv[0] = a; fv[1] = b; fv[2] = c; fv[3] = d;
}
%}
%name(delfv4) void free(void *);
```

An OpenGL package

Whew, we're almost done now. The last thing to do is to package up all of our interface files into a single file called `opengl.i`.

```
//
// opengl.i.    SWIG Interface file for OpenGL
%module opengl

%include gl.i         // The main GL functions
```

```
%include glu.i        // The GLU library
%include glaux.i      // The aux library
%include help.i       // Our helper functions
```

To build the module, we can simply run SWIG as follows :

```
unix > swig -tcl opengl.i          #  Build a dynamicly loaded extension
```

or

```
unix > swig -tcl -lwish.i opengl.i  # Build a statically linked wish executable
```

Compile the file `opengl_wrap.c` with your C compiler and link with Tcl, Tk, and OpenGL to create the final module.

Using the OpenGL module

The module is now used by writing a Tcl script such as the following :

```
load ./opengl.so
auxInitDisplayMode [expr {$AUX_SINGLE | $AUX_RGBA | $AUX_DEPTH}]
auxInitPosition 0 0 500 500
auxInitWindow "Lit-Sphere"

# set up material properties
set mat_specular [newfv4 1.0 1.0 1.0 1.0]
set mat_shininess [newfv4 50.0 0 0 0]
set light_position [newfv4 1.0 1.0 1.0 0.0]

glMaterialfv $GL_FRONT $GL_SPECULAR $mat_specular
glMaterialfv $GL_FRONT $GL_SHININESS $mat_shininess
glLightfv $GL_LIGHT0 $GL_POSITION $light_position
glEnable $GL_LIGHTING
glEnable $GL_LIGHT0
glDepthFunc $GL_LEQUAL
glEnable $GL_DEPTH_TEST

# Set up view
glClearColor 0 0 0 0
glColor3f 1.0 1.0 1.0
glMatrixMode $GL_PROJECTION
glLoadIdentity
glOrtho -1 1 -1 1 -1 1
glMatrixMode $GL_MODELVIEW
glLoadIdentity

# Draw it
glClear $GL_COLOR_BUFFER_BIT
glClear $GL_DEPTH_BUFFER_BIT
auxSolidSphere 0.5

# Clean up
delfv4 $mat_specular
delfv4 $mat_shininess
delfv4 $light_position
```

In our interpreted interface, it is possible to interactively change parameters and see the effects of various OpenGL functions. This a great way to figure out what various functions do and to try different things without having to recompile and run after each change.

Problems with the OpenGL interface

While the OpenGL interface we have generated is fully usable, it is not without problems.

- OpenGL constants are installed as global variables. As a result, it is necessary to use the global keyword when writing Tcl subroutines. For example :

```
proc clear_screan { } {
       global GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT
       glClear $GL_COLOR_BUFFER_BIT
       glClear $GL_DEPTH_BUFFER_BIT
}
```

- Arrays need to be accessed via helper functions such as our `newfv4`() function. This approach certainly works and its easy enough to implement, but it may be preferable to call certain OpenGL functions with a Tcl list instead. For example :

```
glMaterialfv $GL_FRONT $GL_SPECULAR {1.0 1.0 1.0 1.0}
```

While these are only minor annoyances, it turns out that you can address both problems using SWIG typemaps (which are discussed shortly).

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).

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);
}

// 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 :

```
// 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);
            }
    }
```

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.

Tcl 7.x Numerical Conversion Functions

`int Tcl_GetInt(Tcl_Interp *,char *, int *ip)`	Convert a string to an integer which is stored in `ip`. Returns TCL_OK on success, TCL_ERROR on failure.
`int Tcl_GetDouble(Tcl_Interp *, char *, double *dp)`	Convert a string to a double which is stored in `*dp`. Returns TCL_OK on success, TCL_ERROR on failure.
`Tcl_PrintDouble(Tcl_Interp *, double val, char *dest)`	Creates a string with a double precision value. The precision is determined by the value of the `tcl_precision` variable.

Tcl 7.x String and List Manipulation Functions

`void Tcl_SetResult(Tcl_Interp *, char *str, Tcl_FreeProc *freeProc)`	Set the Tcl result string. `str` is the string and `freeProc` is a procedure to free the result. This is usually TCL_STATIC, TCL_DYNAMIC, TCL_VOLATILE.
`void Tcl_AppendResult(Tcl_Interp *, char *str, char *str, ... (char *) NULL)`	Append string elements to the Tcl result string.

Tcl 7.x String and List Manipulation Functions

`void Tcl_AppendElement(Tcl_Interp *,`
`     char *string)`
	Formats string as a Tcl list and appends it to the result string.

`int Tcl_SplitList(Tcl_Interp *, char *list,`
`     int *argcPtr, char ***argvPtr)`
	Parses list as a Tcl list and creates an array of strings. The number of elements is stored in *`argcPtr`. A pointer to the string array is stored in ***`argvPtr`. Returns TCL_OK on success, TCL_ERROR if an error occurred. The pointer value stored in `argvPtr` must eventual be passed to `free`().

`char *Tcl_Merge(int argc, char **argv)`	The inverse of SplitList. Returns a pointer to a Tcl list that has been formed from the array `argv`. The result is dynamically allocated and must be passed to free by the caller.

Tcl 8.x Integer Conversion Functions

`Tcl_Obj *Tcl_NewIntObj(int Value)`	Create a new integer object.
`void Tcl_SetIntObj(Tcl_Obj *obj, int Value)`	Set the value of an integer object

`int Tcl_GetIntFromObj(Tcl_Interp *,`
`                Tcl_Obj *obj, int *ip)`
	Get the integer value of an object and return it in *`ip`. Returns TCL_ERROR if the object is not an integer.

Tcl 8.x Floating Point Conversion Functions

`Tcl_Obj *Tcl_NewDoubleObj(double value)`	Create a new Tcl object containing a double.
`Tcl_SetDoubleObj(Tcl_Obj *obj, double value)`	Set the value of a `Tcl_Object`.

`int Tcl_GetDoubleFromObj(Tcl_Interp,`
`                Tcl_Obj *o, double *dp)`
	Get a double from a Tcl object. The value is stored in *`dp`. Returns TCL_OK on success, TCL_ERROR if the conversion can't be made.

Tcl 8.x String Conversion Functions

`Tcl_Obj *Tcl_NewStringObj(char *str,`
`                int len)`
	Creates a new Tcl string object. `str` contains the ASCII string, `len` contains the number of bytes or -1 if the string is NULL terminated.

`Tcl_SetStringObj(Tcl_Obj *obj, char *str,`
`                int len)`
	Sets a Tcl object to a given string. `len` is the string length or -1 if the string is NULL terminated.

Tcl 8.x String Conversion Functions

`char *Tcl_GetStringFromObj(Tcl_Obj *obj,`
`int *len)`
	Retrieves the string associated with an object. The length is returned in `*len`;

`Tcl_AppendToObj(Tcl_Obj *obj, char *str,`
`int len)`
	Appends the string `str` to the given Tcl Object. `len` contains the number of bytes of -1 if NULL terminated.

Tcl 8.x List Conversion Functions

`Tcl_Obj *Tcl_NewListObj(int objc,`
`Tcl_Obj *objv)`
	Creates a new Tcl List object. `objc` contains the element count and `objv` is an array of Tcl objects.

`int Tcl_ListObjAppendList(Tcl_Interp *,`
`Tcl_Obj *listPtr,`
`Tcl_Obj *elemListPtr)`
	Appends the objects in `elemListPtr` to the list object `listPtr`. Returns TCL_ERROR if an error occurred.

`int Tcl_ListObjAppendElement(Tcl_Interp *,`
`Tcl_Obj *listPtr,`
`Tcl_Obj *element)`
	Appends element to the end of the list object `listPtr`. Returns TCL_ERROR if an error occurred. Will convert the object pointed to by listPtr to a list if it isn't one already.

`int Tcl_ListObjGetElements(Tcl_Interp *,`
`Tcl_Obj *listPtr,`
`int *objcPtr,`
`Tcl_Obj ***objvPtr)`
	Converst a Tcl List object into an array of pointers to individual elements. `objcPtr` receives the list length and `objvPtr` receives a pointer to an array of Tcl_Obj pointers. Returns TCL_ERROR if the list can not be converted.

`int Tcl_ListObjLength(Tcl_Interp *,`
`Tcl_Obj *listPtr,`
`int *intPtr)`
	Returns the length of a list in `intPtr`.If the object is not a list or an error occurs, the function returns TCL_ERROR.

`int Tcl_ListObjIndex(Tcl_Interp *,`
`Tcl_Obj *listPtr,`
`int index, Tcl_Obj **objptr)`
	Returns the pointer to object with given index in the list. Returns TCL_ERROR if `listPtr` is not a list or the index is out of range. The pointer is returned in `objptr`.

`int Tcl_ListObjReplace(Tcl_Interp *,`
`Tcl_Obj *listPtr,`
`int first, int count,`
`int objc, Tcl_Obj *objv)`
	Replaces objects in a list. `first` is the first object to replace and `count` is the total number of objects. `objc` and `objv` define a set of new objects to insert into the list. If `objv` is NULL, no new objects will be added and the function acts as a deletion operation.

Tcl 8.x Object Manipulation

`Tcl_Obj *Tcl_NewObj()`	Create a new Tcl object
`Tcl_Obj *Tcl_DuplicateObj(Tcl_Obj *obj)`	Duplicate a Tcl object.
`Tcl_IncrRefCount(Tcl_Obj *obj)`	Increase the reference count on an object.
`Tcl_DecrRefCount(Tcl_Obj *obj)`	Decrement the reference count on an object.
`int Tcl_IsShared(Tcl_Obj *obj)`	Tests to see if an object is shared.

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)

Function argument typemaps (Tcl 7.x)

`int,`
`short,`
`long`
	`%typemap(tcl,in) int,short,long {`
	`   int temp;`
	`   if (Tcl_GetInt(interp,$source,&temp) == TCL_ERROR)`
	`       return TCL_ERROR;`
	`   $target = ($type) temp;`
	`}`

`float,`
`double`
	`%typemap(tcl,in) double,float {`
	`    double temp;`
	`    if (Tcl_GetDouble(interp,$source,&temp)`
	`        == TCL_ERROR)`
	`        return TCL_ERROR;`
	`    $target = ($type) temp;`
	`}`

`char *`	`%typemap(tcl,in) char * {`
	`  $target = $source;`
	`}`

Function return typemaps (Tcl 7.x)

`int,`
`short,`
`long,`
	`%typemap(tcl,out) int, short, long {`
	`   sprintf($target,”%ld”, (long) $source);`
	`}`

`float,`
`double`
	`%typemap(tcl,out) float,double {`
	`    Tcl_PrintDouble(interp,$source,interp->result);`
	`}`

Function return typemaps (Tcl 7.x)

`char *`	`%typemap(tcl,out) char * {`
	`    Tcl_SetResult(interp,$source,TCL_VOLATILE);`
	`}`

Function argument typemaps (Tcl 8.x)

`int,`
`short,`
`long`
	`%typemap(tcl8,in) int,short,long {`
	`   int temp;`
	`   if (Tcl_GetIntFromObj(interp,$source,&temp) ==`
	`          TCL_ERROR)`
	`       return TCL_ERROR;`
	`   $target = ($type) temp;`
	`}`

`float,`
`double`
	`%typemap(tcl8,in) double,float {`
	`    double temp;`
	`    if (Tcl_GetDoubleFromObj(interp,$source,&temp)`
	`        == TCL_ERROR)`
	`        return TCL_ERROR;`
	`    $target = ($type) temp;`
	`}`

`char *`	`%typemap(tcl8,in) char * {`
	`  int len;`
	`  $target = Tcl_GetStringFromObj(interp,&len);`
	`}`

Function return typemaps (Tcl 8.x)

`int,`
`short,`
`long,`
	`%typemap(tcl8,out) int, short, long {`
	`   Tcl_SetIntObj($target,$source);`
	`}`

`float,`
`double`
	`%typemap(tcl8,out) float,double {`
	`    Tcl_SetDoubleObj($target,$source);`
	`}`

`char *`	`%typemap(tcl8,out) char * {`
	`    Tcl_SetStringObj($target,$source)`
	`}`

Pointer handling

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 .

SWIG Pointer Conversion Functions (Tcl 7.x/8.x)

`void SWIG_MakePtr(char *str, void *ptr, char *type)`
 `void SWIG_SetPointerObj(Tcl_Obj *objPtr, void *ptr, char *type)`
	Makes a pointer string and saves it in `str`, which must be large enough to hold the result. `ptr` contains the pointer value and `type` is the string representation of the type.

`char *SWIG_GetPtr(char *str, void **ptr, char *type)`
 `char *SWIG_GetPointerObj(Tcl_Interp *interp, Tcl_Obj *objPtr, void **ptr, char *_t)`
	Attempts to read a pointer from the string `str`. `ptr` is the address of the pointer to be created and `type` is the expected type. If `type` is NULL, then any pointer value will be accepted. On success, this function returns NULL. On failure, it returns the pointer to the invalid portion of the pointer string.

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 alot 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 Tcl7.5 and newer versions its 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

%}
```

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
extern "C" {
#endif
extern int Tix_Init(Tcl_Interp *);
#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 varibale is overwritten or unset
    upvar $handle_r handle
    #
    # Create the new object
    #
    eval set handle \[new_$objectType $args\]
    #
    # Set up the object procedure
    #
    proc $handle {cmd args} "eval ${objectType}_\$cmd $handle \$args"
    #
    # And the trace ...
    #
    uplevel trace variable $handle_r uw "{deleteObject $objectType $handle}"
    #
    # Return the handle so that 'new' can be used as an argument to a procedure
    #
    return $handle
}

proc deleteObject {objectType handle name element op} {
    #
    # Check that the object handle has a reasonable form
    #
    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.

Extending the Tcl Netscape Plugin

SWIG can be used to extend the Tcl Netscape plugin with C functions. As of this writing it has only been tested with version 1.0 of the plugin on Solaris and Irix 6.2. It may work on other machines as well. However, first a word of caution --- doing this might result in serious injury as you can add just about any C function you want. Furthermore, it's not portable (hey, we're talking C code here). It seems like the best application of this would be creating a browser interface to a highly specialized application. Any scripts that you would write would not work on other machines unless they also installed the C extension code as well. Perhaps we should call this a plugin-plugin...

To use the plugin, use the `-plugin` option :

```
swig -tcl -plugin interface.i
```

This adds a "safe" initialization function compatible with the plugin (in reality, it just calls the function SWIG already creates). You also need to put the following symbol in your interface file for it to work :

```
%{
#define SAFE_SWIG
%}
```

The folks at Sun are quite concerned about the security implications of this sort of extension and originally wanted the user to modify the wrapper code by hand to "remind" them that they were installing functions into a safe interpreter. However, having seen alot of SWIG generated wrapper code, I hated that idea (okay, so the output of SWIG is just a little messy). This is compromise--you need to put that `#define` into your C file someplace. You can also just make it a compiler option if you would like.

The step-by-step process for making a plugin extension.

Making a plugin extension is relatively straightforward but you need to follow these steps :

- Make sure you have Tcl7.6/Tk4.2 installed on your machine. We're going to need the header files into order to compile the extension.
- Make sure you have the Netscape plugin properly installed.
- Run SWIG using the '`-tcl -plugin`' options.
- Compile the extension using the Tcl 7.6/Tk4.2 header files, but linking against the plugin itself. For example :

```
unix > gcc -I/usr/local/include -c example.o interface_wrap.c
unix > ld -shared example.o interface_wrap.o \
      -L/home/beazley/.netscape/plugins/libtclplugin.so -o example.so
```

- Copy the shared object file to the `~/.tclplug/tcl7.7` directory.

Using the plugin

To use the plugin, place the following line in your Tcl scripts :

```
load $tcl_library/example.so example
```

With luck, you will now be ready to run (at least that's the theory).

Tcl8.0 features

SWIG 1.1 now supports Tcl 8.0. However, considering the beta release nature of Tcl 8.0, anything presented here is subject to change. Currently only Tcl 8.0b1 is supported. None of the alpha releases are supported due to a change in the C API.

The Tcl 8.0 module uses the new Tcl 8.0 object interface whenever possible. Instead of using strings, the object interface provides more direct access to objects in their native representation. As a result, the performance is significantly better. The older Tcl SWIG module is also compatible with Tcl 8.0, but since it uses strings it will be much slower than the new version.

In addition to using native Tcl objects, the Tcl8.0 manipulates pointers directly in in a special Tcl object. On the surface it still looks like a string, but internally its represented a (value,type) pair. This too, should offer somewhat better performance.

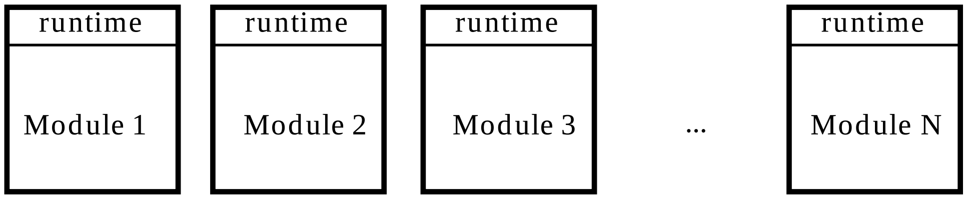
Advanced Topics 11

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 runtime. 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 :



SWIG used with multiple modules (default behavior)

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 :

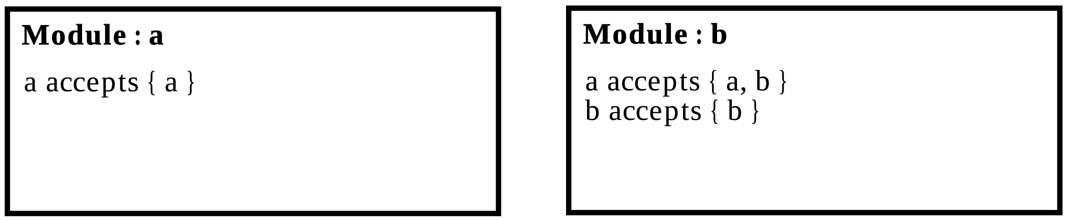
```
// File : a.i
%module a

// Here is a base class
class a {
public:
       a();
       ~a();
       void foo(double);
};
```

```
// File : b.i
%module b

// 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 :



Two SWIG modules with type information

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.

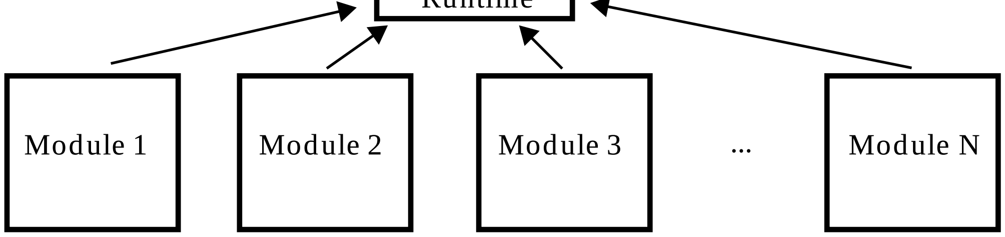
3. Build SWIG modules by linking against the appropriate runtime libraries.

```
% 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, you 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 libary 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 libary at run-time. To find shared libaries, 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, and 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 alot 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 interchangable 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.

Extending SWIG 12

Introduction

This chapter attempts to describe the process of extending SWIG to support new target languages and documentation methods. First a word of warning--SWIG started out being a relatively simple system for building interfaces to ANSI C programs. Since then, it has grown into something much more than that (although I'm still trying to figure out what). As a result, it is undergoing a number of growing pains. Certain parts of the code have been rewritten and others can probably be described as a hackish nightmare. I'm always working on ways to improve the implementation, but expect to find a few warts and inconsistencies.

Prerequisites

In order to develop or modify a SWIG module, I assume the following :

- That you understand the C API for the scripting language of interest.
- You have a good understanding of how SWIG operates and a good idea of how typemaps work.
- That you have some experience with C++. SWIG is written in C++, but doesn't use it maximally. However, familiarity with classes, inheritance, and operator overloading will help.
- That you're just a little crazy (this will help alot).

SWIG Organization

SWIG is built around a central core of functions and classes responsible for parsing interface files, managing documentation, handling datatypes, and utility functions. This code is contained in the "SWIG" directory of the distribution, but contains no information specific to any one scripting language. The various scripting language modules are implemented as C++ classes and found in the "Modules" directory of the distribution. The basic idea behind writing a module is that you write a language class containing about a dozen methods for creating wrapper functions, variables, constants, etc.... To use the language, you simply create a language "object", pass it on the parser and you magically get wrapper code. Documentation modules are written in a similar manner.

An important aspect of the design is the relationship between ANSI C and C++. The original version of SWIG was developed to support ANSI C programs. To add C++ support, an additional "layer" was added to the system---that is, all of the C++ support is really built on top of the ANSI C support. Language modules can take advantage of both C and C++ although a module written only for C can still work with C++ (due to the layered implementation).

As for making modifications to SWIG, all files in the “SWIG” directory should be considered “critical.” Making changes here can cause serious problems in all SWIG language modules. When making customizations, one should only consider files in the “Modules” directory if at all possible.

The organization of this chapter

The remainder of this chapter is a bottom-up approach is to building SWIG modules. It will start with the basics and gradually build up a working language module, introducing new concepts as needed.

Compiling a SWIG extension

The first order of business is that of compiling an extension to SWIG and using it. This is the easy part.

Required files

To build any SWIG extension you need to locate the files “`swig.h`” and “`libswig.a`”. In a typical installation, these will usually be found in `/usr/local/include` and `/usr/local/lib` respectively. All extension modules will need to include the “`swig.h`” header file and link against the `libswig.a` library.

Required C++ compiler

Due to name-mangling in the C++ compiler (which is different between compiler implementations), you will need to use the same C++ compiler used to compile SWIG. If you don’t know which C++ compiler was used, typing ‘`swig -version`’ will cause SWIG to print out its version number and the C++ compiler that was used to build it.

Writing a main program

To get any extension to work, it is necessary to write a small `main()` program to create a language object and start the SWIG parser. For example :

```
#include <swig.h>
#include “swigtcl.h”                      // Language specific header

extern int SWIG_main(int, char **, Language *, Documentation *);
int main(int argc, char **argv) {

       TCL *l = new Tcl;                  // Create a new Language object
       init_args(argc, argv);             // Initialize args
       return SWIG_main(argc, argv, l, 0);
}
```

Compiling

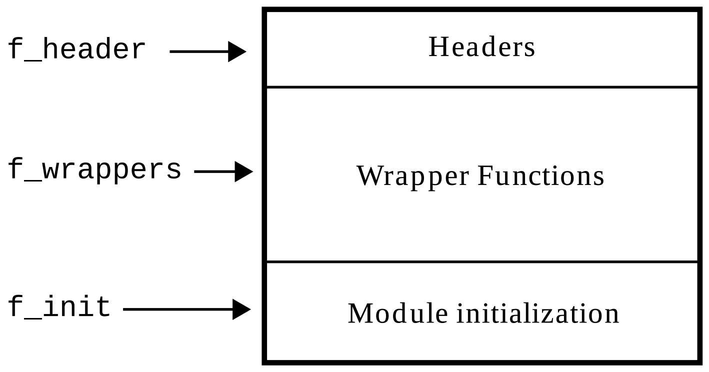
To compile your extension, do the following :

```
% c++ tcl.cxx main.cxx -lswig -o myswig
```

In this case we get a special version of SWIG that compiles Tcl extensions.

SWIG output

The output of SWIG is a single file that is organized as follows :



During code generation, the three sections are created as separate files that are accessed using the following file handles :

```
FILE *f_header;             // Header section
FILE *f_wrappers;           // Wrapper section
FILE *f_init;               // Initialization function
```

On exit, the three files are merged into a single output file.

When generating code, your language module should use the I/O functions in the C `<stdio.h>` library. SWIG does not use the C++ streams library.

The use of each output section can be roughly described as follows :

- The header section contains forward declarations, header files, helper functions, and run-time functions (such as the pointer type-checker). All code included with %{,%} also ends up here.
- The wrapper section contains all of the SWIG generated wrapper functions.
- The initialization section is a single C function used to initialize the module. For large modules, this function can be quite large. In any case, output to `f_init` should be treated with some care considering that the file is essentially one big C function.

The Language class (simple version)

Writing a new language module involves inheriting from the SWIG `Language` class and implementing methods for a few virtual functions. A minimal definition of a new Language module is as follows :

```
// File : mylang.h
// A minimal SWIG Language module

class MYLANG : public Language {
private:
       char *module;
public :
       MYLANG() {
              module = 0;
       };
       // Virtual functions required by the SWIG parser
        void parse_args(int, char *argv[]);
        void parse();
        void create_function(char *, char *, DataType *, ParmList *);
        void link_variable(char *, char *, DataType *);
        void declare_const(char *, char *, DataType *, char *);
        void initialize(void);
        void headers(void);
        void close(void);
        void set_module(char *,char **);
        void create_command(char *, char *);
};
```

Given the above header file, we can create a very simplistic language module as follows :

```
// ---------------------------------------------------------------------
// A simple SWIG Language module
// ---------------------------------------------------------------------

#include "swig.h"
#include "mylang.h"

// ---------------------------------------------------------------------
// MYLANG::parse_args(int argc, char *argv[])
//
// Parse command line options and initializes variables.
// ---------------------------------------------------------------------
void MYLANG::parse_args(int argc, char *argv[]) {
       printf(“Getting command line options\n”);
       typemap_lang = “mylang”;
}

// ---------------------------------------------------------------------
// void MYLANG::parse()
//
// Start parsing an interface file.
// ---------------------------------------------------------------------
void MYLANG::parse() {
       fprintf(stderr,"Making wrappers for My Language\n");
       headers();
       yyparse();       // Run the SWIG parser
}

// ---------------------------------------------------------------------
// MYLANG::set_module(char *mod_name,char **mod_list)
//
// Sets the module name.  Does nothing if it's already set (so it can
// be overridden as a command line option).
//
// mod_list is a NULL-terminated list of additional modules.  This
// is really only useful when building static executables.
//----------------------------------------------------------------------
```

```
void MYLANG::set_module(char *mod_name, char **mod_list) {
  if (module) return;
  module = new char[strlen(mod_name)+1];
  strcpy(module,mod_name);
}

// ----------------------------------------------------------------------
// MYLANG::headers(void)
//
// Generate the appropriate header files for MYLANG interface.
// ----------------------------------------------------------------------
void MYLANG::headers(void) {
       emit_banner(f_header);       // Print the SWIG banner message
       fprintf(f_header,"/* Implementation : My Language */\n\n");
}

// ---------------------------------------------------------------------
// MYLANG::initialize(void)
//
// Produces an initialization function.   Assumes that the module
// name has already been specified.
// ---------------------------------------------------------------------
void MYLANG::initialize() {
       if (!module) module = “swig”;         // Pick a default name
       // Start generating the initialization function
       fprintf(f_init,"int %s_initialize() {\n", module);
}

// ---------------------------------------------------------------------
// MYLANG::close(void)
//
// Finish the initialization function. Close any additional files and
// resources in use.
// ---------------------------------------------------------------------
void MYLANG::close(void) {
       // Finish off our init function
       fprintf(f_init,"}\n");
}

// ---------------------------------------------------------------------
// MYLANG::create_command(char *cname, char *iname)
//
// Creates a new command from a C function.
//            cname = Name of the C function
//            iname = Name of function in scripting language
// ---------------------------------------------------------------------
void MYLANG::create_command(char *cname, char *iname) {
       fprintf(f_init,”\t Creating command %s\n”, iname);
}

// ---------------------------------------------------------------------
// MYLANG::create_function(char *name, char *iname, DataType *d, ParmList *l)
//
// Create a function declaration and register it with the interpreter.
//            name = Name of real C function
//            iname = Name of function in scripting language
//            d = Return datatype
//            l = Function parameters
// ---------------------------------------------------------------------
```

```
void MYLANG::create_function(char *name, char *iname, DataType *d, ParmList *l) {
       fprintf(f_wrappers,"\nwrap_%s() { }\n\n", name);
       create_command(name,iname);
}


// ---------------------------------------------------------------------
// MYLANG::link_variable(char *name, char *iname, DataType *t)
//
// Create a link to a C variable.
//             name = Name of C variable
//             iname = Name of variable in scripting language
//             t = Datatype of the variable
// ---------------------------------------------------------------------
void MYLANG::link_variable(char *name, char *iname, DataType *t) {
       fprintf(f_init,"\t Linking variable : %s\n", iname);
}

// ---------------------------------------------------------------------
// MYLANG::declare_const(char *name, char *iname, DataType *type, char *value)
//
// Makes a constant.
//             name = Name of the constant
//             iname = Scripting language name of constant
//             type = Datatype of the constant
//             value = Constant value (as a string)
// ---------------------------------------------------------------------
void MYLANG::declare_const(char *name, char *iname, DataType *type, char *value) {
       fprintf(f_init,"\t Creating constant : %s = %s\n", name, value);
}
```

To compile our new language, we write a main program (as described previously) and do this :

```
% g++ main.cxx mylang.cxx -I/usr/local/include -L/usr/local/lib -lswig -o myswig
```

Now, try running this new version of SWIG on a few interface files to see what happens. The various `printf()` statements will show you where output appears and how it is structured. For example, if we run this module on the following interface file :

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
%}

// A function
extern double foo(double a, double b);

// A variable
extern int bar;

// A constant
#define SPAM 42
```

We get the following output :

```
/*
```

```
 * FILE : example_wrap.c
 *
 * This file was automatically generated by :
 * Simplified Wrapper and Interface Generator (SWIG)
 * Version 1.1  (Final)
 *
 * Portions Copyright (c) 1995-1997
 * The University of Utah and The Regents of the University of California.
 * Permission is granted to distribute this file in any manner provided
 * this notice remains intact.
 *
 * Do not make changes to this file--changes will be lost!
 *
 */


#define SWIGCODE
/* Implementation : My Language */


/* Put headers and other declarations here */
extern double foo(double ,double );
extern int  bar;

wrap_foo() { }

int example_initialize() {
         Creating command foo
         Linking variable : bar
         Creating constant : SPAM = 42
}
```

Looking at the language module and the output gives some idea of how things are structured. The first part of the file is a banner message printed by the `emit_banner()` function. The "extern" declarations are automatically supplied by the SWIG compiler when use an extern modifier. The wrapper functions appear after all of the headers and forward declarations. Finally, the initialization function is written.

It is important to note that this minimal module is enough to use virtually all aspects of SWIG. If we feed SWIG a C++ file, we will see our low-level module functions being called even though we have not explicitly defined any C++ handling (this is due to the layered approach of implementing C++ on top of C). For example, the following interface file

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

produces accessor functions and the following output :

```
/*
 * FILE : example_wrap.c
 *
 * This file was automatically generated by :
 * Simplified Wrapper and Interface Generator (SWIG)
 * Version 1.1  (Final)
 *
 * Portions Copyright (c) 1995-1997
 * The University of Utah and The Regents of the University of California.
 * Permission is granted to distribute this file in any manner provided
 * this notice remains intact.
 *
 * Do not make changes to this file--changes will be lost!
 *
 */


#define SWIGCODE
/* Implementation : My Language */

static double  Vector_x_set(Vector *obj, double  val) {
    obj->x = val;
    return val;
}

wrap_Vector_x_set() { }

static double  Vector_x_get(Vector *obj) {
    double  result;
    result = (double ) obj->x;
    return result;
}

wrap_Vector_x_get() { }

static double  Vector_y_set(Vector *obj, double  val) {
    obj->y = val;
    return val;
}

wrap_Vector_y_set() { }

static double  Vector_y_get(Vector *obj) {
    double  result;
    result = (double ) obj->y;
    return result;
}

wrap_Vector_y_get() { }

static double  Vector_z_set(Vector *obj, double  val) {
    obj->z = val;
    return val;
}

wrap_Vector_z_set() { }

static double  Vector_z_get(Vector *obj) {
    double  result;
    result = (double ) obj->z;
    return result;
}
```

```
wrap_Vector_z_get() { }

static Vector *new_Vector() {
    return new Vector();
}

wrap_new_Vector() { }

static void delete_Vector(Vector *obj) {
    delete obj;
}

wrap_delete_Vector() { }

static double  Vector_magnitude(Vector *obj) {
    double  _result = (double )obj->magnitude();
    return _result;
}

wrap_Vector_magnitude() { }

int example_initialize() {
         Creating command Vector_x_set
         Creating command Vector_x_get
         Creating command Vector_y_set
         Creating command Vector_y_get
         Creating command Vector_z_set
         Creating command Vector_z_get
         Creating command new_Vector
         Creating command delete_Vector
         Creating command Vector_magnitude
}
```

With just a little work, we already see that SWIG does quite alot for us. Now our task is to fill in the various Language methods with the real code needed to produce a working module. Before doing that, we first need to take a tour of some important SWIG datatypes and functions.

A tour of SWIG datatypes

While SWIG has become somewhat complicated over the last year, its internal operation is based on just a few fundamental datatypes. These types are described now although examples of using the various datatypes are shown later.

The DataType class

All C datatypes are represented by the following structure :

```
class DataType {
public:
       DataType();
       DataType(DataType *);
       ~DataType();
       int         type;               // SWIG Type code
       char        name[MAXNAME];      // Name of type
```

```
        char       is_pointer;          // Is this a pointer?
        char       implicit_ptr;        // Implicit ptr
        char       is_reference;        // A C++ reference type
        char       status;              // Is this datatype read-only?
        char       *qualifier;          // A qualifier string (ie. const).
        char       *arraystr;           // String containing array part
        int        id;                  // type identifier (unique for every type).
        // Output methods
        char      *print_type();        // Return string containing datatype
        char      *print_full();        // Return string with full datatype
        char      *print_cast();        // Return string for type casting
        char      *print_mangle();      // Return mangled version of type
        char      *print_mangle_default(); // Default mangling scheme
        char      *print_real();        // Print the real datatype
        char      *print_arraycast();   // Prints an array cast
        // Array query functions
        int       array_dimensions();   // Return number of array dimensions (if any)
        char      *get_dimension(int);  // Return string for a particular dimension
};
```

The fundamental C datatypes are given a unique numerical code which is stored in the `type` field. The current list of types is as follows :

C Datatype	SWIG Type Code
int	T_INT
short	T_SHORT
long	T_LONG
char	T_CHAR
float	T_FLOAT
double	T_DOUBLE
void	T_VOID
unsigned int	T_UINT
unsigned short	T_USHORT
unsigned long	T_ULONG
unsigned char	T_UCHAR
signed char	T_SCHAR
bool	T_BOOL
<user>	T_USER
error	T_ERROR

The `T_USER` type is used for all derived datatypes including structures and classes. The `T_ERROR` type indicates that a parse/type error has occurred and went undetected (as far as I know this doesn't happen).

The `name[]` field contains the actual name of the datatype as seen by the parser and is currently limited to a maximum of 96 bytes (more than enough for most applications). If a typedef has been used, the name field contains the actual name used, not the name of the primitive C datatype. Here are some examples :

C Datatype	type	name[]
double	T_DOUBLE	double
unsigned int	T_UINT	unsigned int
signed long	T_LONG	signed long
struct Vector	T_USER	struct Vector
Real	T_DOUBLE	Real

C qualifiers such as "`const`" or "`volatile`" are stored separately in the `qualifier` field. In order to produce usable wrapper code, SWIG often needs to strip the qualifiers. For example, trying to assign a passed function argument into a type of "`const int`" will irritate most compilers. Unfortunately, this kind of assignment is unavoidable when converting arguments between a scripting and C representation.

The `is_pointer` field indicates whether or not a particular datatype is a pointer. The value of `is_pointer` determines the level of indirection used. For example :

C Datatype	**type**	**is pointer**
double *	T_DOUBLE	1
int ***	T_INT	3
char *	T_CHAR	1

The `implicit_ptr` field is an internally used parameter that is used to properly handle the use of pointers in typedef statements. However, for the curious, in indicates the level of indirection implicitly defined in a datatype. For example :

```
typedef char *String;
```

is represented by a datatype with the following parameters :

```
type          = T_CHAR;
name[]        = "String";
is_pointer    = 1;
implicit_ptr  = 1;
```

Normally, language modules do not worry about the `implicit_ptr` field.

C++ references are indicated by the `is_reference` field. By default, the parser converts references into pointers which makes them indistinguishable from other pointer datatypes. However, knowing that something is a reference effects some code generation procedures so this field can be checked to see if a datatype really is a C++ reference.

The `arraystr` field is used to hold the array dimensions of array datatypes. The dimensions are simply represented by a string. For example :

C Datatype	type	is pointer	arraystr
double a[50]	T_DOUBLE	1	[50]
int b[20][30][50]	T_INT	1	[20][30][50]
char *[MAX]	T_CHAR	2	[MAX]

SWIG converts all arrays into pointers. Thus a "`double [50]`" is really just a special version of "`double *`". If a datatype is not declared as an array, the `arraystr` field contains the NULL pointer.

A collection of "output" methods are available for datatypes. The names of these methods are mainly "historical" in that they don't actually "print" anything, but now return character strings instead. Assuming that `t` is a datatype representing the C datatype "`const int *`", here's what the methods produce :

Operation	**Output**
t->print_type()	int *
t->print_full()	const int *
t->print_cast()	(int *)
t->print_mangle()	< language dependent >
t->print_mangle_default()	_int_p

A few additional output methods are provided for dealing with arrays :

type	**Operation**	**Output**
int a[50]	t->print_type()	int *
int a[50]	t->print_real()	int [50]
int a[50]	t->print_arraycast()	(int *)
int a[50][50]	t->print_arraycast()	(int (*)[50])

Additional information about arrays is also available using the following functions :

type	**Operation**	**Result**
int a[50]	t->array_dimension()	1
int a[50]	t->get_dimension(0)	50
int b[MAXN][10]	t->array_dimension()	2
int b[MAXN][10]	t->get_dimension(0)	MAXN
int b[MAXN][10]	t->get_dimension(1)	10

The `DataType` class contains a variety of other methods for managing typedefs, scoping, and other operations. These are usually only used by the SWIG parser. While available to language modules too, they are never used (at least not in the current implementation), and should probably be avoided unless you absolutely know what you're doing (not that this is a strict requirement of course).

Function Parameters

Each argument of a function call is represented using the `Parm` structure :

```
struct Parm {
       Parm(DataType *type, char *name);
       Parm(Parm *p);
       ~Parm();
       DataType   *t;               // Datatype of this parameter
       int        call_type;        // Call type (value or reference or value)
       char       *name;            // Name of parameter (optional)
       char       *defvalue;        // Default value (as a string)
       int        ignore;           // Ignore flag
};
```

`t` is the datatype of the parameter, `name` is an optional parameter name, and `defvalue` is a default argument value (if supplied).

`call_type` is an integer code describing any special processing. It can be one of two values :

- `CALL_VALUE`. This means that the argument is a pointer, but we should make it work like a call-by-value argument in the scripting interface. This value used to be set by the `%val` directive, but this approach is now deprecated (since the same effect can be achieved by typemaps).

- `CALL_REFERENCE`. This is set when a complex datatype is being passed by value to a function. Since SWIG can't handle complex datatypes by value, the datatype is implicitly changed into a pointer and `call_type` set to `CALL_REFERENCE`. Many of SWIG's internals look at this when generating code. A common mistake is forgetting that all complex datatypes in SWIG are pointers. This is even the case when writing a language-module---the conversion to pointers takes place in the parser before data is even passed into a particular module.

The `ignore` field is set when SWIG detects that a function parameter is to be "ignored" when generating wrapper functions. An "ignored" parameter is usually set to a default value and effectively disappears when a function call is made from a scripting language (that is, the function is called with fewer arguments than are specified in the interface file). The `ignore` field is normally only set when an "ignore" typemap has been used.

All of the function parameters are passed in the structure `ParmList`. This structure has the following user-accesible methods available :

```
class ParmList {
public:
  int   nparms;                  // Number of parms in list
  void  append(Parm *p);         // Append a parameter to the end
  void  insert(Parm *p, int pos); // Insert a parameter into the list
  void  del(int pos);            // Delete a parameter at position pos
  int   numopt();                // Get number of optional arguments
  int   numarg();                // Get number of active arguments
  Parm *get(int pos);            // Get the parameter at position pos
};
```

The methods operate in the manner that you would expect. The most common operation that will be performed in a language module is walking down the parameter list and processing individual parameters. This can be done as follows :

```
// Walk down a parameter list
ParmList *l;                 // Function parameter list (already created)
Parm *p;

for (int i = 0; i < l->nparms; i++) {
       p = l->get(i);              // Get ith parameter
       // do something with the parameter
       ...
}
```

The String Class

The process of writing wrapper functions is mainly just a tedious exercise in string manipulation. To make this easier, the `String` class provides relatively simple mechanism for constructing strings, concatenating values, replacing symbols, and so on.

```
class String {
public:
  String();
  String(const char *s);
  ~String();
  char  *get() const;
  friend String& operator<<(String&,const char *s);
  friend String& operator<<(String&,const int);
  friend String& operator<<(String&,const char);
  friend String& operator<<(String&,String&);
  friend String& operator>>(const char *s, String&);
  friend String& operator>>(String&,String&);
  String& operator=(const char *);
  operator char*() const { return str; }
  void   untabify();
  void   replace(char *token, char *rep);
  void   replaceid(char *id, char *rep);
};
```

Strings can be manipulated in a manner that looks similar to C++ I/O operations. For example :

```
String s;

s << "void" << " foo() {\n"
  << tab4 << "printf(\"Hello World\");\n"
  << "}\n";

fprintf(f_wrappers,"%s", (char *) s);
```

produces the output :

```
void foo() {
    printf("Hello World");
}
```

The << operator always appends to the end of a string while >> can be used to insert a string at the beginning. Strings may be used anywhere a `char *` is expected. For example :

```
String s1,s2;
...
if (strcmp(s1,s2) == 0) {
       printf("Equal!\n");
}
```

The `get()` method can be used to explicitly return the `char *` containing the string data. The `untabify()` method replaces tabs with whitespace. The `replace()` method can be used to perform substring replacement and is used by typemaps. For example :

```
s.replace("$target","_arg3");
```

The `replaceid()` method can be used to replace valid C identifiers with a new value. C identifiers must be surrounded by white-space or other non-identifier characters. The `replace()` method does not have this restriction.

Hash Tables

Hash tables can be created using the `Hash` class :

```
class Hash {
public:
  Hash();
  ~Hash();
  int    add(const char *key, void *object);
  int    add(const char *key, void *object, void (*del)(void *));
  void  *lookup(const char *key);
  void   remove(const char *key);
  void  *first();
  void  *next();
  char  *firstkey();
  char  *nextkey();
};
```

Hash tables store arbitrary objects (cast to `void *`) with string keys. An optional object deletion function may be registered with each entry to delete objects when the hash is destroyed. Hash tables are primarily used for managing internal symbol tables although language modules may also use them to keep track of special symbols and other state.

The following hash table shows how one might keep track of real and renamed function names.

```
Hash wrapped_functions;

int add_function(char *name, char *renamed) {
       char *nn = new char[strlen(renamed)+1];
       strcpy(nn,renamed);
       if (wrapped_functions.add(name,nn) == -1) {
             printf("Function multiply defined!\n");
             delete [] nn;
             return -1;
       }
}
char *get_renamed(char *name) {
       char *rn = (char *) wrapped_functions.lookup(name);
       return rn;
}
```

The `remove()` method removes a hash-table entry. The `first()` and `next()` methods are iterators for extracting all of the hashed objects. They return NULL when no more objects are found in the hash. The `firstkey()` and `nextkey()` methods are iterators that return the hash keys. NULL is returned when no more keys are found.

The WrapperFunction class

Finally, a `WrapperFunction` class is available for simplifying the creation of wrapper functions. The class is primarily designed to organize code generation and provide a few supporting services. The class is defined as follows :

```
class WrapperFunction {
public:
  String  def;
  String  locals;
  String  code;
  void    print(FILE *f);
  void    print(String &f);
  void    add_local(char *type, char *name, char *defvalue = 0);
  char   *new_local(char *type, char *name, char *defvalue = 0);
};
```

Three strings are available. The `def` string contains the actual function declaration, the `locals` string contain local variable declarations, and the `code` string contains the resulting wrapper code.

The method `add_local()` creates a new local variable which is managed with an internal symbol table (that detects variable conflicts and reports potential errors). The `new_local()` method can be used to create a new local variable that is guaranteed to be unique. Since a renaming might be required, this latter method returns the name of the variable that was actually selected for use (typically, this is derived from the original name).

The `print()` method can be used to emit the wrapper function to a file. The printing process consolidates all of the strings into a single result.

Here is a very simple example of the wrapper function class :

```
WrapperFunction f;

f.def << “void count_n(int n) {“;
f.add_local(“int”,”i”);
f.code << tab4 << “for (i = 0; i < n; i++) {\n“
       << tab8 << “printf(\”%d\\n\”,i);\n”
       << tab4 << “}\n”
       << “}\n”;

f.print(f_wrappers);
```

This produces the following output :

```
void count_n(int n) {
    int i;
    for (i = 0; i < n; i++) {
         printf(“%d\n”,i);
    }
}
```

Of course, as you can guess, the functions actually generated by your language module will be more complicated than this.

Typemaps (from C)

The typemapper plays a big role in generating code for all of SWIG’s modules. Understanding the `%typemap` directive and how it works is probably a good starting point for understanding this section.

The typemap C API.

There is a relatively simple C API for managing typemaps in language modules.

```
void typemap_register(char *op, char *lang, DataType *type, char *pname,
                      char *code, ParmList *l = 0);
```

Registers a new typemap with the typemapper. This is the C equivalent of the `%typemap` directive. For example :

```
%typemap(lang,op) type pname { ... code ... };
%typemap(lang,op) type pname(ParmList) { ... code ... };
```

`code` contains the actual typemap code, while `l` is a parameter list containing local variable declarations. Normally, it is not necessary to execute this function from language modules.

`void typemap_register_default(char *op, char *lang, int type, int ptr, char *arraystr, char *code, ParmList *args)`

Registers a default typemap. This works in the same way as the normal registration function, but takes a type code (an integer) instead. Default typemaps are more general than normal typemaps (see below).

`char *typemap_lookup(char *op, char *lang, DataType *type, char *pname, char *source, char *target, WrapperFunction *f = 0);`

Looks up a typemap, performs variable substitutions, and returns a string with the corresponding typemap code. The tuple `(op, lang, type, pname)` determine which typemap to find. `source` contains the string to be assigned to the `$source` variable, `target` contains the string to be assigned to the `$target` variable. `f` is an optional wrapper function object. It should be supplied to support parameterized typemaps (ie. typemaps that declare additional local variables).

`typemap_lookup()` returns NULL is no typemap is found. Otherwise it returns a string containing the updated typemap code. This code has a number of variables substituted including `$source`, `$target`, `$type`, `$mangle`, and `$basetype`.

`char *typemap_check(char *op, char *lang, DataType *type, char *pname)`

Checks to see if a typemap exists. The tuple `(op, lang, type, pname)` determine which typemap to find. If the typemap exists, the raw typemap code is returned. Otherwise, NULL is returned. While `typemap_lookup()` could be used to accomplish the same thing, this function is more compact and is significantly faster since it does not perform any variable substitutions.

What happens on typemap lookup?

When looking for a typemap, SWIG searches for a match in a series of steps.

- Explicit typemaps. These are specified directly with the `%typemap()` directive. Named typemaps have the highest precedence while arrays have higher precedence than pointers (see the typemap chapter for more details).
- If no explicit typemap is found, mappings applied with the `%apply` directive are checked. Basically, `%apply` is nothing more than a glorified renaming operation. We rename the datatype and see if there are any explicit typemaps that match. If so, we use the typemap that was found.
- Default typemaps. If no match is found with either explicit typemaps or apply direc-

tives, we make a final search using the default typemap. Unlike other typemaps, default typemaps are applied to the raw SWIG internal datatypes (`T_INT`, `T_DOUBLE`, `T_CHAR`, etc...). As a result, they are insentive to typedefs and renaming operations. If nothing is found here, a NULL pointer is returned indicating that no mapping was found for that particular datatype.

Another way to think of the typemap mechanism is that it always tries to apply the most specific typemap that can be found for any particular datatype. When searching, it starts with the most specific and works its way out to the most general specification. If nothing is found it gives up and returns a NULL pointer.

How many typemaps are there?

All typemaps are identified by an operation string such as "in", "out", "memberin", etc... A number of typemaps are defined by other parts of SWIG, but you can create any sort of typemap that you wish by simply picking a new name and using it when making calls to `typemap_lookup()` and `typemap_check()`.

File management

The following functions are provided for managing files within SWIG.

`void add_directory(char *dirname);`

Adds a new directory to the search path used to locate SWIG library files. This is the C equivalent of the `swig -I` option.

`int insert_file(char *filename, FILE *output);`

Searches for a file and copies it into the given output stream. The search process goes through the SWIG library mechanism which first checks the current directory, then in various parts of the SWIG library for a match. Returns -1 if the file is not found. Language modules often use this function to insert supporting code. Usually these code fragments are given a '`.swg`' suffix and are placed in the SWIG library.

`int get_file(char *filename, String &str);`

Searches for a file and returns its contents in the String `str.`Returns a -1 if the file is not found.

`int checkout_file(char *source, char *dest);`

Copies a file from the SWIG library into the current directory. `dest` is the filename of the desired file. This function will not replace a file that already exists. The primary use of this function is to give the user supporting code. For example, we could check out a Makefile if none exists. Returns -1 on failure.

`int include_file(char *filename);`

The C equivalent of the SWIG `%include` directive. When called, SWIG will attempt to open `filename` and start parsing all of its contents. If successful, parsing of the new file will take place immediately. When the end of the file is reached, the parser switches back to the input file being read prior to this call. Returns -1 if the file is not found.

Naming Services

The naming module provides methods for generating the names of wrapper functions, accessor functions, and other aspects of SWIG. Each function returns a new name that is a syntactically correct C identifier where invalid characters have been converted to a "_".

`char *name_wrapper(char *fname, char *prefix);`
Returns the name of a wrapper function. By default, it will be "`_wrap_prefixfname`".

`char *name_member(char *mname, char *classname;`
Returns the name of a C++ accessor function. Normally, this is "`classname_mname`".

`char *name_get(char *vname);`
Returns the name of a function to get the value of a variable or class data member. Normally "`vname_get`" is returned.

`char *name_set(char *vname);`
Returns the name of a function to set the value of a variable or class data member. Normally, "`vname_set`" is returned.

`char *name_construct(char *classname);`
Returns the name of a constructor function. Normally returns "`new_classname`".

`char *name_destroy(char *classname);`
Returns the name of a destructor function. Normally returns "`delete_classname`".

Each function may also accept an optional parameter of `AS_IS`. This suppresses the conversion of illegal characters (a process that is sometimes required). For example :

```
char *name = name_member("foo","bar",AS_IS);   // Produce a name, but don't change
                                                // illegal characters.
```

It is critical that language modules use the naming functions. These function are used throughout SWIG and provide a centralized mechanism for keeping track of functions that have been generated, managing multiple files, and so forth. In future releases, it may be possible to change the naming scheme used by SWIG. Using these functions should insure future compatibility.

Code Generation Functions

The following functions are used to emit code that is generally useful and used in essentially every SWIG language module.

`int emit_args(DataType *t, ParmList *l, WrapperFunction &f);`
Creates all of the local variables used for function arguments and return value. `t` is the return datatype of the function, `l` is the parameter list holding all of the function argu-

ments. `f` is a WrapperFunction object where the local variables will be created.

`void emit_func_call(char *name, DataType *t, ParmList *l, WrapperFunction &f);`

Creates a function call to a C function. `name` is the name of the C function, `t` is the return datatype, `l` is the function parameters and `f` is a WrapperFunction object. The code generated by this function assumes that one has first called `emit_args()`.

`void emit_banner(FILE *file);`

Emits the SWIG banner comment to the output file.

`void emit_set_get(char *name, char *rename, DataType *t);`

Given a variable of type `t`, this function creates two functions to set and get the value. These functions are then wrapped like normal C functions. `name` is the real name of the variable. `rename` is the renamed version of the variable.

`void emit_ptr_equivalence(FILE *file);`

To keep track of datatypes, SWIG maintains an internal table of “equivalent” datatypes. This table is updated by typedef, class definitions, and other C constructs. This function emits code compatible with the type-checker that is needed to make sure pointers work correctly. Typically this function is called after an interface file has been parsed completely.

Writing a Real Language Module

Whew, assuming you’ve made it this far, we’re ready to write a real language module. In this example, we’ll develop a simple Tcl module. Tcl has been chosen because it has a relatively simple C API that is well documented and easy to understand. The module developed here is not the same as the real SWIG Tcl module (which is significantly more complicated).

The header file

We will start with the same header file as before :

```
// File : mylang.h
// A simple SWIG Language module

class MYLANG : public Language {
private:
       char *module;
public :
       MYLANG() {
              module = 0;
       };
       // Virtual functions required by the SWIG parser
        void parse_args(int, char *argv[]);
        void parse();
        void create_function(char *, char *, DataType *, ParmList *);
        void link_variable(char *, char *, DataType *);
        void declare_const(char *, char *, DataType *, char *);
        void initialize(void);
        void headers(void);
        void close(void);
```

```
            void set_module(char *,char **);
            void create_command(char *, char *);
    };
```

Command Line Options and Basic Initialization

Command line options are parsed using the `parse_args()` method :

```
    // ------------------------------------------------------------------------
    // A simple SWIG Language module
    //
    // ------------------------------------------------------------------------

    #include "swig.h"
    #include "mylang.h"

    static char *usage = "\
    My Language Options\n\
         -module name    - Set name of module\n\n";

    // ---------------------------------------------------------------------
    // MYLANG::parse_args(int argc, char *argv[])
    //
    // Parse my command line options and initialize by variables.
    // ---------------------------------------------------------------------

    void MYLANG::parse_args(int argc, char *argv[]) {
      // Look for certain command line options
      for (int i = 1; i < argc; i++) {
        if (argv[i]) {
          if (strcmp(argv[i],"-module") == 0) {
            if (argv[i+1]) {
              set_module(argv[i+1],0);
              mark_arg(i);
              mark_arg(i+1);
              i++;
            } else {
              arg_error();
            }
          } else if (strcmp(argv[i],"-help") == 0) {
            fprintf(stderr,"%s\n", usage);
          }
        }
      }
      // Set location of SWIG library
      strcpy(LibDir,"tcl");

      // Add a symbol to the parser for conditional compilation
      add_symbol("SWIGTCL",0,0);

      // Add typemap definitions
      typemap_lang = "tcl”;
    }
```

Parsing command line options follows the same conventions as for writing a C++ main program with one caveat. For each option that your language module parses, you need to call the function `mark_arg()`. This tells the SWIG main program that your module found a valid option and used it. If you don't do this, SWIG will exit with an error message about unrecognized command line options.

After processing command line options, you next need to set the variable `LibDir` with the name of the subdirectory your language will use to find files in the SWIG library. Since we are making a new Tcl module, we'll just set this to "`tcl`".

Next, we may want to add a symbol to SWIG's symbol table. In this case we're adding "`SWIGTCL`" to indicate that we're using Tcl. SWIG modules can use this for conditional compilation and detecting your module using `#ifdef`.

Finally, we need to set the variable `typemap_lang`. This should be assigned a name that you would like to use for all typemap declarations. When a user gives a typemap, they would use this name as the target language.

Starting the parser

To start the SWIG parser, the `parse()` method is used :

```
// ------------------------------------------------------------------
// void MYLANG::parse()
//
// Start parsing an interface file for MYLANG.
// ------------------------------------------------------------------

void MYLANG::parse() {

       fprintf(stderr,"Making wrappers for Tcl\n");
       headers();       // Emit header files and other supporting code

       // Tell the parser to first include a typemap definition file

       if (include_file("lang.map") == -1) {
           fprintf(stderr,"Unable to find lang.map!\n");
           SWIG_exit(1);
       }
       yyparse();       // Run the SWIG parser
}
```

This function should print some kind of message to the user indicating what language is being targeted. The `headers()` method is called (see below) to emit support code and header files. Finally, we make a call to `yyparse()`. This starts the SWIG parser and does not return until the entire interface file has been read.

In our implementation, we have also added code to immediately include a file '`lang.map`'. This file will contain typemap definitions to be used by our module and is described in detail later.

Emitting headers and support code

Prior to emitting any code, our module should emit standard header files and support code. This is done using the `headers()` method :

```
// ------------------------------------------------------------------
// MYLANG::headers(void)
//
```

```
// Generate the appropriate header files for MYLANG interface.
// ----------------------------------------------------------------------

void MYLANG::headers(void)
{
  emit_banner(f_header);               // Print the SWIG banner message
  fprintf(f_header,"/* Implementation : My TCL */\n\n");

  // Include header file code fragment into the output
  if (insert_file("header.swg",f_header) == -1) {
    fprintf(stderr,"Fatal Error. Unable to locate 'header.swg'.\n");
    SWIG_exit(1);
  }

  // Emit the default SWIG pointer type-checker (for strings)
  if (insert_file("swigptr.swg",f_header) == -1) {
    fprintf(stderr,"Fatal Error. Unable to locate 'swigptr.swg'.\n");
    SWIG_exit(1);
  }
}
```

In this implementation, we emit the standard SWIG banner followed by a comment indicating which language module is being used. After that, we are going to include two different files. '`header.swg`' is a file containing standard declarations needed to build a Tcl extension. For our example, it will look like this :

```
/* File : header.swg */
#include <tcl.h>
```

The file '`swigptr.swg`' contains the standard SWIG pointer-type checking library. This library contains about 300 lines of rather nasty looking support code that define the following 3 functions :

`void SWIG_RegisterMapping(char *type1, char *type,`
`void *(*cast)(void *))`

Creates a mapping between C datatypes `type1` and `type2`. This is registered with the runtime type-checker and is similiar to a typedef. `cast` is an optional function pointer defining a method for proper pointer conversion (if needed). Normally, cast is only used when converting between base and derived classes in C++ and is needed for proper implementation of multiple inheritance.

`void SWIG_MakePtr(char *str, void *ptr, char *type);`

Makes a string representation of a C pointer. The result is stored in `str` which is assumed to be large enough to hold the result. `ptr` contains the pointer value and `type` is a string code corresponding to the datatype.

`char *SWIG_GetPtr(char *str, void **ptr, char *type);`

Extracts a pointer from its string representation, performs type-checking, and casting. `str` is the string containing the pointer-value representation, `ptr` is the address of the pointer that will be returned, and `type` is the string code corresponding to the datatype. If a type-error occurs, the function returns a `char *` corresponding to the part of the input string that was invalid, otherwise the function returns NULL. If a NULL pointer is given for `type`, the function will accept a pointer of any type.

We will use these functions later.

Setting a module name

The `set_module()` method is used whenever the `%module` directive is encountered.

```
// ---------------------------------------------------------------------
// MYLANG::set_module(char *mod_name,char **mod_list)
//
// Sets the module name.  Does nothing if it's already set (so it can
// be overriddent as a command line option).
//
// mod_list is a NULL-terminated list of additional modules to initialize
// and is only provided if the user specifies something like this :
// %module foo, mod1, mod2, mod3, mod4
//----------------------------------------------------------------------

void MYLANG::set_module(char *mod_name, char **mod_list) {
       if (module) return;
       module = new char[strlen(mod_name)+1];
       strcpy(module,mod_name);
       // Make sure the name conforms to Tcl naming conventions
       for (char *c = module; (*c); c++)
              *c = tolower(*c);
       toupper(module);
}
```

This function may, in fact, be called multiple times in the course of processing. Normally, we only allow a module name to be set once and ignore all subsequent calls however.

Final Initialization

The initialization of a module takes several steps--parsing command line options, printing standard header files, starting the parser, and setting the module name. The final step in initialization is calling the `initialize()` method :

```
// --------------------------------------------------------------------
// MYLANG::initialize(void)
//
// Produces an initialization function.   Assumes that the module
// name has already been specified.
// ---------------------------------------------------------------------

void MYLANG::initialize()
{
       // Check if a module has been defined
       if (!module) {
              fprintf(stderr,"Warning. No module name given!\n");
              module = "swig";
       }

       // Generate a CPP symbol containing the name of the initialization function
       fprintf(f_header,"#define SWIG_init    %s_Init\n\n\n", module);

       // Start generating the initialization function
       fprintf(f_init,"int SWIG_init(Tcl_Interp *interp) {\n");
```

```
	fprintf(f_init,"\t if (interp == 0) return TCL_ERROR;\n");
}
```

The `initialize()` method should create the module initialization function by emitting code to `f_init` as shown. By this point, we should already know the name of the module, but should check just in case. The preferred style of creating the initialization function is to create a C preprocessor symbol `SWIG_init`. Doing so may look weird, but it turns out that many SWIG library files may want to know the name of the initialization function. If we define a symbol for it, these files can simply assume that it's called `SWIG_init()` and everything will work out (okay, so it's a hack).

Cleanup

When an interface file has finished parsing, we need to clean everything up. This is done using the `close()` method :

```
// -----------------------------------------------------------------------
// MYLANG::close(void)
//
// Wrap things up.  Close initialization function.
// -----------------------------------------------------------------------

void MYLANG::close(void)
{
  // Dump the pointer equivalency table
  emit_ptr_equivalence(f_init);

  // Finish off our init function and print it to the init file
  fprintf(f_init,"\t return TCL_OK;\n");
  fprintf(f_init,"}\n");
}
```

The `close()` method should first call `emit_ptr_equivalence()` if the SWIG pointer type checker has been used. This dumps out support code to make sure the type-checker works correctly. Afterwards, we simply need to terminate our initialization function as shown. After this function has been called, SWIG dumps out all of its documentation files and exits.

Creating Commands

Now, we're moving into the code generation part of SWIG. The first step is to make a function to create scripting language commands. This is done using the `create_command()` function :

```
// -----------------------------------------------------------------------
// MYLANG::create_command(char *cname, char *iname)
//
// Creates a Tcl command from a C function.
// -----------------------------------------------------------------------
void MYLANG::create_command(char *cname, char *iname) {
	// Create a name for the wrapper function
	char *wname = name_wrapper(cname,"");

	// Create a Tcl command
	fprintf(f_init,"\t Tcl_CreateCommand(interp,\"%s\",%s, (ClientData) NULL,
		(Tcl_CmdDeleteProc *) NULL);\n", iname, wname);
}
```

For our Tcl module, this just calls Tcl_CreateCommand to make a new scripting language command.

Creating a Wrapper Function

The most complicated part of writing a language module is the process of creating wrapper functions. This is done using the `create_function()` method as shown here :

```
// ------------------------------------------------------------------------
// MYLANG::create_function(char *name, char *iname, DataType *d, ParmList *l)
//
// Create a function declaration and register it with the interpreter.
// ------------------------------------------------------------------------

void MYLANG::create_function(char *name, char *iname, DataType *t, ParmList *l)
{
  String           source, target;
  char             *tm;
  String           cleanup, outarg;
  WrapperFunction  f;

  // Make a wrapper name for this function
  char *wname = name_wrapper(iname,"");

  // Now write the wrapper function itself
  f.def << "static int " << wname << "(ClientData clientData, Tcl_Interp *interp, int
argc, char *argv[]) {\n";

  // Emit all of the local variables for holding arguments.
  int pcount = emit_args(t,l,f);

  // Get number of optional/default arguments
  int numopt = l->numopt();

  // Emit count to check the number of arguments
  f.code << tab4 << "if ((argc < " << (pcount-numopt) + 1 << ") || (argc > "
         << l->numarg()+1 << ")) {\n"
         << tab8 << "Tcl_SetResult(interp, \"Wrong # args.\",TCL_STATIC);\n"
         << tab8 << "return TCL_ERROR;\n"
         << tab4 << "}\n";

  // Now walk the function parameter list and generate code to get arguments
  int j = 0;                // Total number of non-optional arguments

  for (int i = 0; i < pcount ; i++) {
    Parm &p = (*l)[i];         // Get the ith argument
    source = "";
    target = "";

    // Produce string representation of source and target arguments
    source << "argv[" << j+1 << "]";
    target << "_arg" << i;
    if (!p.ignore) {
      if (j >= (pcount-numopt))  // Check if parsing an optional argument
        f.code << tab4 << "if argc >" << j+1 << ") {\n";

      // Get typemap for this argument
      tm = typemap_lookup("in",typemap_lang,p.t,p.name,source,target,&f);
```

```
        if (tm) {
          f.code << tm << "\n";
          f.code.replace("$arg",source);   // Perform a variable replacement
        } else {
          fprintf(stderr,"%s : Line %d. No typemapping for datatype %s\n",
                  input_file,line_number, p.t->print_type());
        }
        if (j >= (pcount-numopt))
          f.code << tab4 << "} \n";
        j++;
      }

      // Check to see if there was any sort of a constaint typemap
      if ((tm = typemap_lookup("check",typemap_lang,p.t,p.name,source,target))) {
        f.code << tm << "\n";
        f.code.replace("$arg",source);
      }

      // Check if there was any cleanup code (save it for later)
      if ((tm = typemap_lookup("freearg",typemap_lang,p.t,p.name,target,
                               "interp->result"))) {
        cleanup << tm << "\n";
        cleanup.replace("$arg",source);
      }
      if ((tm = typemap_lookup("argout",typemap_lang,p.t,p.name,target,
                               "interp->result"))) {
        outarg << tm << "\n";
        outarg.replace("$arg",source);
      }
    }

    // Now write code to make the function call
    emit_func_call(name,t,l,f);

    // Return value if necessary
    if ((t->type != T_VOID) || (t->is_pointer)) {
      if ((tm = typemap_lookup("out",typemap_lang,t,name,"_result","interp->result"))) {
        // Yep.  Use it instead of the default
        f.code << tm << "\n";
      } else {
        fprintf(stderr,"%s : Line %d. No return typemap for datatype %s\n",
                input_file,line_number,t->print_type());
      }
    }

    // Dump argument output code;
    f.code << outarg;

    // Dump the argument cleanup code
    f.code << cleanup;

    // Look for any remaining cleanup.  This processes the %new directive
    if (NewObject) {
      if ((tm = typemap_lookup("newfree",typemap_lang,t,iname,"_result",""))) {
        f.code << tm << "\n";
      }
    }

    // Special processing on return value.
```

```
  if ((tm = typemap_lookup("ret",typemap_lang,t,name,"_result",""))) {
    f.code << tm << "\n";
  }

  // Wrap things up (in a manner of speaking)
  f.code << tab4 << "return TCL_OK;\n}";

  // Substitute the cleanup code (some exception handlers like to have this)
  f.code.replace("$cleanup",cleanup);

  // Emit the function
  f.print(f_wrappers);

  // Now register the function with the language
  create_command(iname,iname);
}
```

Creating a wrapper function really boils down to 3 components :

- Emit local variables and handling input arguments.
- Call the real C function.
- Convert the return value to a scripting language representation.

In our implementation, most of this work is done using typemaps. In fact, the role of the C++ code is really just to process typemaps in the appropriate order and to combine strings in the correct manner. The following typemaps are used in this procedure :

- "in". This is used to convert function arguments from Tcl to C.
- "out". This is used to convert the return value from C to Tcl.
- "check". This is used to apply constraints to the input values.
- "argout". Used to return values through function parameters.
- "freearg". Used to clean up arguments after a function call (possibly to release memory, etc...)
- "ret". Used to clean up the return value of a C function (possibly to release memory).
- "newfree" this is special processing applied when the `%new` directive has been used. Usually its used to clean up memory.

It may take awhile for this function to sink in, but its operation will hopefully become more clear shortly.

Manipulating Global Variables

To provide access to C global variables, the `link_variable()` method is used. In the case of Tcl, only `int`, `double`, and `char *` datatypes can be safely linked.

```
// -----------------------------------------------------------------------
// MYLANG::link_variable(char *name, char *iname, DataType *t)
//
// Create a Tcl link to a C variable.
// -----------------------------------------------------------------------

void MYLANG::link_variable(char *name, char *iname, DataType *t) {
  char *tm;
```

```
    // Uses a typemap to stick code into the module initialization function
    if ((tm = typemap_lookup("varinit",typemap_lang,t,name,name,iname))) {
      String temp = tm;
      if (Status & STAT_READONLY)
        temp.replace("$status"," | TCL_LINK_READ_ONLY");
      else
        temp.replace("$status","");
      fprintf(f_init,"%s\n",(char *) temp);
    } else {
      fprintf(stderr,"%s : Line %d. Unable to link with variable type %s\n",
              input_file,line_number,t->print_type());
    }
  }
```

In this case, the procedure is looking for a typemap "varinit". We'll use the code specified with this typemap to create variable links. If no typemap is supplied or the user gives an unsupported datatypes, a warning message will be generated.

It is also worth noting that the `Status` variable contains information about whether or not a variable is read-only or not. To test for this, use the technique shown in the code above. Read-only variables may require special processing as shown.

Constants

Finally, creating constants is accomplished using the `declare_const()` method. For Tcl, we could do this :

```
  // -------------------------------------------------------------------------
  // MYLANG::declare_const(char *name, char *iname, DataType *type, char *value)
  //
  // Makes a constant.
  // --------------------------------------------------------------------------

  void MYLANG::declare_const(char *name, char *iname, DataType *type, char *value) {

    char *tm;
    if ((tm = typemap_lookup("const",typemap_lang,type,name,name,iname))) {
      String str = tm;
      str.replace("$value",value);
      fprintf(f_init,"%s\n", (char *) str);
    } else {
      fprintf(stderr,"%s : Line %d. Unable to create constant %s = %s\n",
              input_file, line_number, type->print_type(), value);
    }
  }
```

We take the same approach used to create variables. In this case, the '`const`' typemap specifies the special processing.

The value of a constant is a string produced by the SWIG parser. It may contain an arithmetic expression such as "3 + 4*(7+8)". Because of this, it is critical to use this string in a way that allows it to be evaluated by the C compiler (this will be apparent when the typemaps are given).

A Quick Intermission

We are now done writing all of the methods for our language class. Of all of the methods, `create_function()` is the most complicated and tends to do most of the work. We have also ignored issues related to documentation processing and C++ handling (although C++ will work with the functions we have defined so far).

While our C++ implementation is done, we still do not have a working language module. In fact, if we run SWIG on the following interface file :

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
%}

// A function
extern double foo(double a, double b);

// A variable
extern int bar;

// A constant
#define SPAM 42
```

we get the following errors :

```
[beazley@guinness lang]$ ./myswig example.i
Making wrappers for My Tcl
example.i : Line 9. No typemapping for datatype double
example.i : Line 9. No typemapping for datatype double
example.i : Line 9. No return typemap for datatype double
example.i : Line 12. Unable to link with variable type int
example.i : Line 16. Unable to create constant int  = 42
[beazley@guinness lang]$
```

The reason for this is that we have not yet defined any processing for real datatypes. For example, our language module has no idea how to convert doubles into Tcl strings, how to link with C variables and so on. To do this, we need to write a collection of typemaps.

Writing the default typemaps

In our earlier `parse()` method, there is a statement to include the file '`lang.map`'. We will use this file to write typemaps for our new language module. The '`lang.map`' file will actually go through the SWIG parser so we can write our typemaps using the normal `%typemap` directive. This approach makes it easy for us to debug and test our module because the typemaps can be developed and tested without having to repeatedly recompile the C++ part of the module.

Without further delay, here is the typemap file for our module (you might want to sit down) :

```
// ----------------------------------------------------------------------
// lang.map
//
// This file defines all of the type-mappings for our language (TCL).
// A typemap of 'SWIG_DEFAULT_TYPE' should be used to create default
// mappings.
// ----------------------------------------------------------------------
```

```
/**************************** FUNCTION INPUTS ****************************/

// Integers
%typemap(in) int             SWIG_DEFAULT_TYPE,
             short           SWIG_DEFAULT_TYPE,
             long            SWIG_DEFAULT_TYPE,
             unsigned int    SWIG_DEFAULT_TYPE,
             unsigned short  SWIG_DEFAULT_TYPE,
             unsigned long   SWIG_DEFAULT_TYPE,
             signed char     SWIG_DEFAULT_TYPE,
             unsigned char   SWIG_DEFAULT_TYPE
{
  int  temp;
  if (Tcl_GetInt(interp, $source, &temp) == TCL_ERROR) return TCL_ERROR;
  $target = ($type) temp;
}

// Floating point
%typemap(in) float   SWIG_DEFAULT_TYPE,
             double  SWIG_DEFAULT_TYPE
{
  double temp;
  if (Tcl_GetDouble(interp, $source, &temp) == TCL_ERROR) return TCL_ERROR;
  $target = ($type) temp;
}

// Strings
%typemap(in) char *  SWIG_DEFAULT_TYPE
{
  $target = $source;
}

// void *
%typemap(in) void *  SWIG_DEFAULT_TYPE
{
  if (SWIG_GetPtr($source,(void **) &$target, (char *) 0)) {
    Tcl_SetResult(interp,"Type error.  Expected a pointer",TCL_STATIC);
    return TCL_ERROR;
  }
}

// User defined types and all other pointers
%typemap(in) User * SWIG_DEFAULT_TYPE
{
  if (SWIG_GetPtr($source,(void **) &$target, "$mangle")) {
    Tcl_SetResult(interp,"Type error.  Expected a $mangle",TCL_STATIC);
    return TCL_ERROR;
  }
}

/**************************** FUNCTION OUTPUTS ****************************/

// Signed integers
%typemap(out) int         SWIG_DEFAULT_TYPE,
              short       SWIG_DEFAULT_TYPE,
              long        SWIG_DEFAULT_TYPE,
              signed char SWIG_DEFAULT_TYPE
{
  sprintf($target,"%ld", (long) $source);
```

```
}

// Unsigned integers
%typemap(out) unsigned       SWIG_DEFAULT_TYPE,
              unsigned short SWIG_DEFAULT_TYPE,
              unsigned long  SWIG_DEFAULT_TYPE,
              unsigned char  SWIG_DEFAULT_TYPE
{
  sprintf($target,"%lu", (unsigned long) $source);
}

// Floating point
%typemap(out) double SWIG_DEFAULT_TYPE,
              float  SWIG_DEFAULT_TYPE
{
  Tcl_PrintDouble(interp,(double) $source,interp->result);
}

// Strings
%typemap(out) char *SWIG_DEFAULT_TYPE
{
  Tcl_SetResult(interp,$source,TCL_VOLATILE);
}

// Pointers
%typemap(out) User *SWIG_DEFAULT_TYPE
{
  SWIG_MakePtr($target,(void *) $source, "$mangle");
}

/*************************** VARIABLE CREATION ***************************/

// Integers
%typemap(varinit) int          SWIG_DEFAULT_TYPE,
                  unsigned int SWIG_DEFAULT_TYPE
{
  Tcl_LinkVar(interp, "$target", (char *) &$source, TCL_LINK_INT $status);
}

// Doubles
%typemap(varinit) double SWIG_DEFAULT_TYPE {
  Tcl_LinkVar(interp,"$target", (char *) &$source, TCL_LINK_DOUBLE $status);
}

// Strings
%typemap(varinit) char * SWIG_DEFAULT_TYPE {
  Tcl_LinkVar(interp,"$target", (char *) &$source, TCL_LINK_STRING $status);
}

/***************************** CONSTANTS *********************************/

// Signed Integers
%typemap(const) int           SWIG_DEFAULT_TYPE,
                short         SWIG_DEFAULT_TYPE,
                long          SWIG_DEFAULT_TYPE,
                signed char   SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(40);
  sprintf(_wrap_$target,"%ld",$value);
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Unsigned integers
%typemap(const) unsigned       SWIG_DEFAULT_TYPE,
                unsigned short  SWIG_DEFAULT_TYPE,
                unsigned long   SWIG_DEFAULT_TYPE,
                unsigned char   SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(40);
  sprintf(_wrap_$target,"%lu",$value);
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Doubles and floats
%typemap(const) double SWIG_DEFAULT_TYPE,
                float  SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(40);
  sprintf(_wrap_$target,"%f",$value);
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Strings
%typemap(const) char *SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target = "$value";
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Pointers
%typemap(const) User *SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(20+strlen("$mangle"));
  SWIG_MakePtr(_wrap_$target, (void *) $value, "$mangle");
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}
```

Now that we have our typemaps file, we are done and can start producing a variety of interesting Tcl extension modules. Should errors arrise, one will either have to pry into the C++ module or the typemaps file for a correction.

The SWIG library and installation issues

To make a new SWIG module generally usable, you will want to perform the following steps :

- Put the new binary in a publicly accessible location (ie. `/usr/local/bin`).
- Make a subdirectory for your language in the SWIG library. The library should match up

with the name you assigned to the `LibDir` variable in `parse_args()`.
- Copy the file '`lang.map`' to the SWIG library directory. Your new version of SWIG will now be able to find it no matter what directory SWIG is executed from.
- Provide some documentation about how your module works.

SWIG extensions are only able to target a single scripting language. If you would like to make your module part of the full version of SWIG, you will need to modify the file '`swigmain.cxx`' in the `SWIG1.1/Modules` directory. To do this, follow these steps :

- Add a `#include "lang.h"` to the `swigmain.cxx` file.
- Create a command line option for your module and write code to create an instance of your language (just copy the technique used for the other languages).
- Modify `Modules/Makefile` to include your module as part of its compilation process.
- Rebuild SWIG by typing 'make'.

C++ Processing

Language modules have the option to provide special processing for C++ classes. Usually this is to provide some sort of object-oriented interface such as shadow classes. The process of developing these extensions is highly technical and the best approach may be to copy pieces from other SWIG modules that provide object oriented support.

How C++ processing works

The wrapping of C++ classes follows a "file" metaphor. When a class is encountered, the following steps are performed :

- Open a new class.
- Inherit from base classes.
- Add members to the class (functions, variables, constants, etc...)
- Close the class and emit object-oriented code.

As a class is constructed, a language module may need to keep track of a variety of data such as whether constructors or destructors have been given, are there any data members, have datatypes been renamed, and so on. It is not always a clear-cut process.

Language extensions

Providing additional support for object-oriented programming requires the use of the following Language extensions. These are additional methods that can be defined for the Language class.

`void cpp_open_class(char *name, char *rename, char *ctype, int strip);`

Opens a new class. `name` is the name of the class, `rename` is the renamed version of the class (or NULL if not renamed), `ctype` is the class type (`struct`, `class`, `union`), and `strip` is a flag indicating whether or not its safe to drop the leading type specifier (this is often unsafe for ANSI C).

`void cpp_inherit(char **baseclass, int mode = INHERIT_ALL);`

Inherits from base classes. `baseclass` is a NULL terminated array of class names corresponding to all of the base classes of an object. `mode` is an inheritance mode that is the or'd value of `INHERIT_FUNC` , `INHERIT_VAR`, `INHERIT_CONST`, or `INHERIT_ALL`.

```
void cpp_member_func(char *name, char *iname, DataType *t,
                     ParmList *l);
```

Creates a member function. `name` is the real name of the member, `iname` is the renamed version (NULL if not renamed), `t` is the return datatype, and `l` is the function parameter list.

```
void cpp_static_func(char *name, char *iname, DataType *t,
                     ParmList *l);
```

Create a static member function. The calling conventions are the same as for `cpp_member_func()`.

```
void cpp_variable(char *name, char *iname, DataType *t);
```

Creates a member variable. `name` is the real name of the member, `iname` is the renamed version (NULL is not renamed). `t` is the type of the member.

```
void cpp_static_var(char *name, char *iname, DataType *t);
```

Creates a static member variable. The calling convention is the same as for `cpp_variable()`.

```
void cpp_declare_const(char *name, char *iname, DataType *type,
                       char *value);
```

Creates a constant inside a C++ class. Normally this is an enum or member declared as const. `name` is the real name, `iname` is the renamed version (NULL if not renamed), `type` is the type of the constant, and `value` is a string containing the value.

```
void cpp_constructor(char *name, char *iname, ParmList *l);
```

Creates a constructor. `name` is the name of the constructor, `iname` is the renamed version, and `l` is the function parameter list. Normally, `name` is the same name as the class. If not, this may actually be a member function with no declared return type (assumed to be an int in C++).

```
void cpp_destructor(char *name, char *newname);
```

Creates a destructor. `name` is the real name of the destructor (usually the same name as the class), and `newname` is the renamed version (NULL if not renamed).

```
void cpp_close_class();
```

Closes the current class. Language modules should finish off code generation for a class once this has been called.

```
void cpp_cleanup();
```

Called after all C++ classes have been generated. Only used to provide some kind of global cleanup for all classes.

Hints

Special C++ code generation is not for the weak of heart. Most of SWIG's built in modules have been developed for well over a year and object oriented support has been in continual development. If writing a new language module, looking at the implementation for Python, Tcl, or Perl5 would be a good start.

Documentation Processing

The documentation system operates (for the most part), independently of the language modules. However, language modules are still responsible for generating a “usage” string describing how each function, variable, or constant is to be used in the target language.

Documentation entries

Each C/C++ declaration in an interface file gets assigned to a “Documentation Entry” that is described by the `DocEntry` object :

```
class DocEntry {
public:
  String      usage;            // Short description
  String      cinfo;            // Information about C interface (optional).
  String      text;             // Supporting text (optional)
};
```

The `usage` string is used to hold the calling sequence for the function. The `cinfo` field is used to provide additional information about the underlying C code. `text` is filled in with comment text.

The global variable `doc_entry` always contains the documentation entry for the current declaration being processed. Language modules can choose to update the documentation by referring to and modifying its fields.

Creating a usage string

To create a documentation usage string, language modules need to modify the ‘usage’ field of `doc_entry`. This can be done by creating a function like this :

```
// ---------------------------------------------------------------------------
// char *TCL::usage_string(char *iname, DataType *t, ParmList *l),
//
// Generates a generic usage string for a Tcl function.
// ---------------------------------------------------------------------------

char * TCL::usage_string(char *iname, DataType *, ParmList *l) {

  static String temp;
  Parm  *p;
  int   i, numopt,pcount;

  temp = "";
  temp << iname << " ";

  /* Now go through and print parameters */
  i = 0;
  pcount = l->nparms;
  numopt = l->numopt();
  p = l->get_first();
  while (p != 0) {
    if (!p->ignore) {
      if (i >= (pcount-numopt))
        temp << "?";

      /* If parameter has been named, use that.   Otherwise, just print a type  */

      if ((p->t->type != T_VOID) || (p->t->is_pointer)) {
        if (strlen(p->name) > 0) {
          temp << p->name;
        }
        else {
          temp << "{ " << p->t->print_type() << " }";
        }
      }
      if (i >= (pcount-numopt))
        temp << "?";
      temp << " ";
      i++;
    }
    p = l->get_next();
  }
  return temp;
}
```

Now, within the function to create a wrapper function, include code such as the following :

```
// Fill in the documentation entry
doc_entry->usage << usage_string(iname,t,l);
```

To produce full documentation, each language module needs to fill in the documentation usage string for all declarations. Looking at existing SWIG modules can provide more information on how this should be implemented.

Writing a new documentation module

Writing a new documentation module is roughly the same idea as writing a new Language class. To do it, you need to implement a “Documentation Object” by inheriting from the following base class and defining all of the virtual methods :

```
class Documentation {
public:
  virtual void parse_args(int argc, char **argv) = 0;
  virtual void title(DocEntry *de) = 0;
  virtual void newsection(DocEntry *de, int sectnum) = 0;
  virtual void endsection() = 0;
  virtual void print_decl(DocEntry *de) = 0;
  virtual void print_text(DocEntry *de) = 0;
  virtual void separator() = 0;
  virtual void init(char *filename) = 0;
  virtual void close(void) = 0;
  virtual void style(char *name, char *value) = 0;
};
```

`void parse_args(int argc, char **argv);`

Parses command line options. Any special options you want to provide should be placed here.

void title(DocEntry *de;
Produces documentation output for a title.

void newsection(DocEntry *de, int sectnum;
Called whenever a new section is created. The documentation system is hierarchical in nature so each call to this function goes down one level in the hierarchy.

void endsection();
Ends the current section. Moves up one level in the documentation hierarchy.

void print_decl(DocEntry *de);
Creates documentation for a C declaration (function, variable, or constant).

void print_text(DocEntry *de);
Prints documentation that has been given with the `%text %{ %}` directive.

void separator();
Prints an optional separator between sections.

void init(char *filename);
Initializes the documentation system. This function is called after command line options have been parsed. `filename` is the file where documentation should be placed.

void close(void);
Closes the documentation file. All remaining output should be complete and files closed upon exit.

void style(char *name, char *value);
Called to set style parameters. This function is called by the `%style` and `%localstyle` directives. It is also called whenever style parameters are given after a section directive.

Using a new documentation module

Using a new documentation module requires a change to SWIG's main program. If you are writing your own main() program, you can use a new documentation module as follows :

```
#include <swig.h>
#include "swigtcl.h"                        // Language specific header
#include "mydoc.h"                          // New Documentation module

extern int SWIG_main(int, char **, Language *, Documentation *);
int main(int argc, char **argv) {

       TCL *l = new TCL;                    // Create a new Language object
       MyDoc *d = new MyDoc;                // New documentation object
       init_args(argc, argv);               // Initialize args
       return SWIG_main(argc, argv, l, d);
}
```

Where to go for more information

To find out more about documentation modules, look at some of the existing SWIG modules contained in the `SWIG1.1/SWIG` directory. The ASCII and HTML modules are good starting points for finding more information.

The Future of SWIG

SWIG's C++ API is the most rapidly evolving portion of SWIG. While interface-file compatibility will be maintained as much as possible in future releases, the internal structure of SWIG is likely to change significantly in the future. This will possibly have many ramifications on the construction of language modules. Here are a few significant changes that are coming :

1. A complete reorganization of the SWIG type system. Datatypes will be represented in a more flexible manner that provide full support for arrays, function pointers, classes, structures, and types possibly coming from other languages (such as Fortran).
2. Objectification of functions, variables, constants, classes, etc... Currently many of SWIG's functions take multiple arguments such as functions being described by name, return datatype, and parameters. These attributes will be consolidated into a single "Function" object, "Variable" object, "Constant" object, and so forth.
3. A complete rewrite of the SWIG parser. This will be closely tied to the change in datatype representation among other things.
4. Increased reliance on typemaps. Many of SWIG's older modules do not rely on typemaps, but this is likely to change. Typemaps provide a more concise implementation and are easier to maintain. Modules written for 1.1 that adopt typemaps now will be much easier to integrate into future releases.
5. A possible reorganization of object-oriented code generation. The layered approach will probably remain in place however.
6. Better support for multiple-files (exporting, importing, etc...)

In planning for the future, much of a language's functionality can be described in terms of typemaps. Sticking to this approach will make it significantly easier to move to new releases. I anticipate that there will be few drastic changes to the Language module presented in this section (other than changes to many of the calling mechanisms).

Index

(Index is still under construction)