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 dec-

laration 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
```

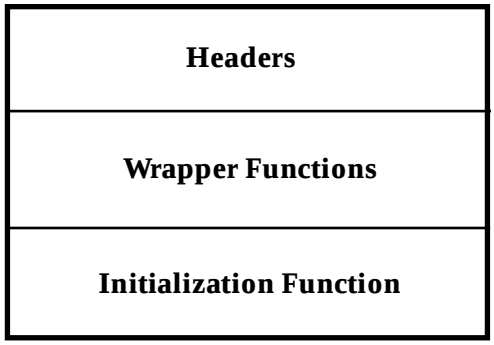
Interface files can look at these symbols as necessary to change the way in which an interface is generated or to mix SWIG directives with C code. These symbols are also defined within the C code generated by SWIG (except for the symbol '`SWIG`' which is only defined within the SWIG compiler).

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).