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 val-

ues 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.

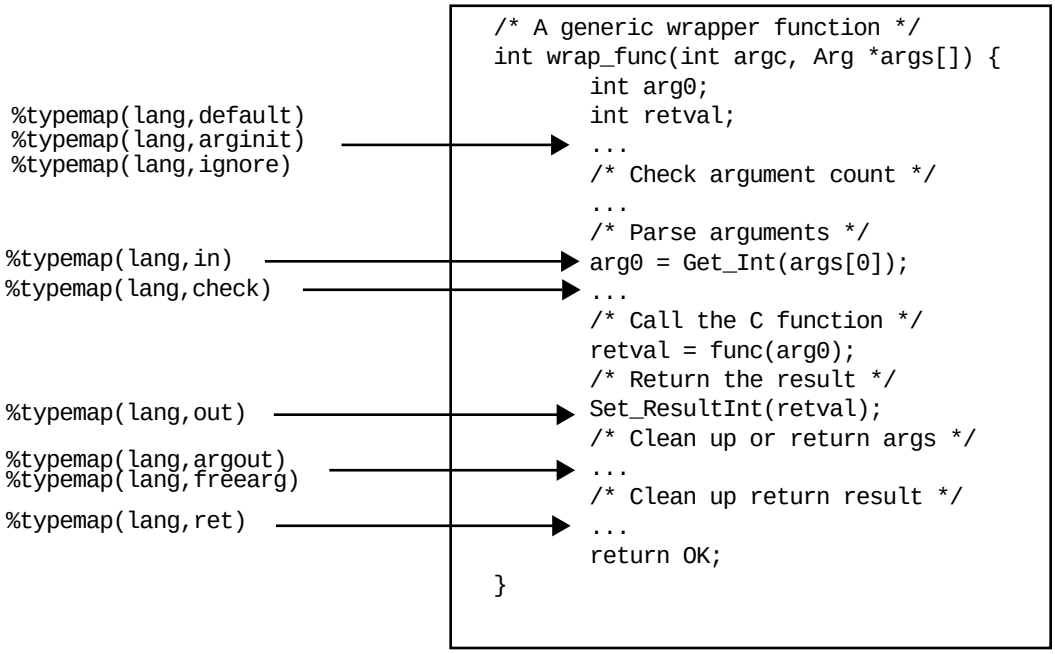
Common Typemap Methods

`%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.

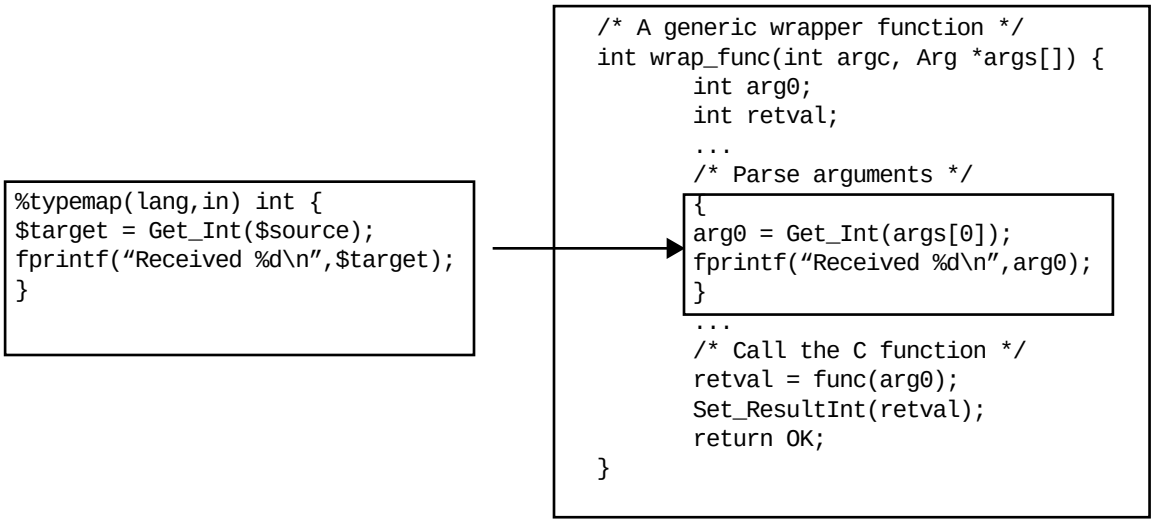
Common Typemap Methods

`%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.



Wrapper function typemaps and where they are applied



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 con-

straints. 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.