10 Typemaps

10.1 Introduction

Chances are, you are reading this chapter for one of two reasons; you either want to customize SWIG's behavior or you overheard someone mumbling some incomprehensible drivel about "typemaps" and you asked yourself "typemaps, what are those?" That said, let's start with a short disclaimer that "typemaps" are an advanced customization feature that provide direct access to SWIG's low-level code generator. Not only that, they are an integral part of the SWIG C++ type system (a non-trivial topic of its own). Typemaps are generally *not* a required part of using SWIG. Therefore, you might want to re-read the earlier chapters if you have found your way to this chapter with only a vague idea of what SWIG already does by default.

10.1.1 Type conversion

One of the most important problems in wrapper code generation is the conversion of datatypes between programming languages. Specifically, for every C/C++ declaration, SWIG must somehow generate wrapper code that allows values to be passed back and forth between languages. Since every programming language represents data differently, this is not a simple of matter of simply linking code together with the C linker. Instead, SWIG has to know something about how data is represented in each language and how it can be manipulated.

To illustrate, suppose you had a simple C function like this:

```
int factorial(int n);
```

To access this function from Python, a pair of Python API functions are used to convert integer values. For example:

```
long PyInt_AsLong(PyObject *obj);      /* Python --> C */
PyObject *PyInt_FromLong(long x);      /* C --> Python */
```

The first function is used to convert the input argument from a Python integer object to C `long`. The second function is used to convert a value from C back into a Python integer object.

Inside the wrapper function, you might see these functions used like this:

```
PyObject *wrap_factorial(PyObject *self, PyObject *args) {
    int       arg1;
    int       result;
    PyObject *obj1;
    PyObject *resultobj;

    if (!PyArg_ParseTuple("O:factorial", &obj1)) return NULL;
    arg1 = PyInt_AsLong(obj1);
    result = factorial(arg1);
    resultobj = PyInt_FromLong(result);
    return resultobj;
}
```

Every target language supported by SWIG has functions that work in a similar manner. For example, in Perl, the following functions are used:

```
IV SvIV(SV *sv);                    /* Perl --> C */
void sv_setiv(SV *sv, IV val);      /* C --> Perl */
```

In Tcl:

```
int Tcl_GetLongFromObj(Tcl_Interp *interp, Tcl_Obj *obj, long *value);
Tcl_Obj *Tcl_NewIntObj(long value);
```

The precise details are not so important. What is important is that all of the underlying type conversion is handled by collections of utility functions and short bits of C code like this---you simply have to read the extension documentation for your favorite language to know how it works (an exercise left to the reader).

10.1.2 Typemaps

Since type handling is so central to wrapper code generation, SWIG allows it to be completely defined (or redefined) by the user. To do this, a special `%typemap` directive is used. For example:

```
/* Convert from Python --> C */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}

/* Convert from C --> Python */
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

At first glance, this code will look a little confusing. However, there is really not much to it. The first typemap (the "in" typemap) is used to convert a value from the target language to C. The second typemap (the "out" typemap) is used to convert in the other direction. The content of each typemap is a small fragment of code that is inserted directly into the SWIG generated wrapper functions. The code is usually C or C++ code which will be generated into the C/C++ wrapper functions. Note that this isn't always the case as some target language modules allow target language code within the typemaps which gets generated into target language specific files. Within this code, a number of special variables prefixed with a $ are expanded. These are really just placeholders for C/C++ variables that are generated in the course of creating the wrapper function. In this case, `$input` refers to an input object that needs to be converted to C/C++ and `$result` refers to an object that is going to be returned by a wrapper function. `$1` refers to a C/C++ variable that has the same type as specified in the typemap declaration (an `int` in this example).

A short example might make this a little more clear. If you were wrapping a function like this:

```
int gcd(int x, int y);
```

A wrapper function would look approximately like this:

```
PyObject *wrap_gcd(PyObject *self, PyObject *args) {
   int arg1;
   int arg2;
   int result;
   PyObject *obj1;
   PyObject *obj2;
   PyObject *resultobj;

   if (!PyArg_ParseTuple("OO:gcd", &obj1, &obj2)) return NULL;

   /* "in" typemap, argument 1 */
   {
```

```
      arg1 = PyInt_AsLong(obj1);
   }

   /* "in" typemap, argument 2 */
   {
      arg2 = PyInt_AsLong(obj2);
   }

   result = gcd(arg1,arg2);

   /* "out" typemap, return value */
   {
      resultobj = PyInt_FromLong(result);
   }

   return resultobj;
}
```

In this code, you can see how the typemap code has been inserted into the function. You can also see how the special $ variables have been expanded to match certain variable names inside the wrapper function. This is really the whole idea behind typemaps--they simply let you insert arbitrary code into different parts of the generated wrapper functions. Because arbitrary code can be inserted, it possible to completely change the way in which values are converted.

10.1.3 Pattern matching

As the name implies, the purpose of a typemap is to "map" C datatypes to types in the target language. Once a typemap is defined for a C datatype, it is applied to all future occurrences of that type in the input file. For example:

```
/* Convert from Perl --> C */
%typemap(in) int {
   $1 = SvIV($input);
}

...
int factorial(int n);
int gcd(int x, int y);
int count(char *s, char *t, int max);
```

The matching of typemaps to C datatypes is more than a simple textual match. In fact, typemaps are fully built into the underlying type system. Therefore, typemaps are unaffected by `typedef`, namespaces, and other declarations that might hide the underlying type. For example, you could have code like this:

```
/* Convert from Ruby--> C */
%typemap(in) int {
   $1 = NUM2INT($input);
}
...
typedef int Integer;
namespace foo {
    typedef Integer Number;
};

int foo(int x);
int bar(Integer y);
int spam(foo::Number a, foo::Number b);
```

In this case, the typemap is still applied to the proper arguments even though typenames don't always match the text "int". This ability to track types is a critical part of SWIG--in fact, all of the target language modules work merely define a set of typemaps for the basic types. Yet, it

is never necessary to write new typemaps for typenames introduced by `typedef`.

In addition to tracking typenames, typemaps may also be specialized to match against a specific argument name. For example, you could write a typemap like this:

```
%typemap(in) double nonnegative {
   $1 = PyFloat_AsDouble($input);
   if ($1 < 0) {
        PyErr_SetString(PyExc_ValueError,"argument must be nonnegative.");
        return NULL;
   }
}

...
double sin(double x);
double cos(double x);
double sqrt(double nonnegative);

typedef double Real;
double log(Real nonnegative);
...
```

For certain tasks such as input argument conversion, typemaps can be defined for sequences of consecutive arguments. For example:

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

In this case, a single input object is expanded into a pair of C arguments. This example also provides a hint to the unusual variable naming scheme involving `$1`, `$2`, and so forth.

10.1.4 Reusing typemaps

Typemaps are normally defined for specific type and argument name patterns. However, typemaps can also be copied and reused. One way to do this is to use assignment like this:

```
%typemap(in) Integer = int;
%typemap(in) (char *buffer, int size) = (char *str, int len);
```

A more general form of copying is found in the `%apply` directive like this:

```
%typemap(in) int {
   /* Convert an integer argument */
   ...
}
%typemap(out) int {
   /* Return an integer value */
   ...
}

/* Apply all of the integer typemaps to size_t */
%apply int { size_t };
```

`%apply` merely takes *all* of the typemaps that are defined for one type and applies them to other types. Note: you can include a comma separated set of types in the `{ ... }` part of `%apply`.

It should be noted that it is not necessary to copy typemaps for types that are related by `typedef`. For example, if you have this,

```
typedef int size_t;
```

then SWIG already knows that the `int` typemaps apply. You don't have to do anything.

10.1.5 What can be done with typemaps?

The primary use of typemaps is for defining wrapper generation behavior at the level of individual C/C++ datatypes. There are currently six general categories of problems that typemaps address:

Argument handling

```
int foo(int x, double y, char *s);
```

- Input argument conversion ("in" typemap).
- Input argument type checking ("typecheck" typemap).
- Output argument handling ("argout" typemap).
- Input argument value checking ("check" typemap).
- Input argument initialization ("arginit" typemap).
- Default arguments ("default" typemap).
- Input argument resource management ("freearg" typemap).

Return value handling

```
int foo(int x, double y, char *s);
```

- Function return value conversion ("out" typemap).
- Return value resource management ("ret" typemap).
- Resource management for newly allocated objects ("newfree" typemap).

Exception handling

```
int foo(int x, double y, char *s) throw(MemoryError, IndexError);
```

- Handling of C++ exception specifications. ("throw" typemap).

Global variables

```
int foo;
```

- Assignment of a global variable. ("varin" typemap).
- Reading a global variable. ("varout" typemap).

Member variables

```
struct Foo {
    int x[20];
};
```

- Assignment of data to a class/structure member. ("memberin" typemap).

Constant creation

```
#define FOO 3
%constant int BAR = 42;
enum { ALE, LAGER, STOUT };
```

- Creation of constant values. ("consttab" or "constcode" typemap).

Details of each of these typemaps will be covered shortly. Also, certain language modules may define additional typemaps that expand upon this list. For example, the Java module defines a variety of typemaps for controlling additional aspects of the Java bindings. Consult language specific documentation for further details.

10.1.6 What can't be done with typemaps?

Typemaps can't be used to define properties that apply to C/C++ declarations as a whole. For example, suppose you had a declaration like this,

```
Foo *make_Foo();
```

and you wanted to tell SWIG that `make_Foo()` returned a newly allocated object (for the purposes of providing better memory management). Clearly, this property of `make_Foo()` is *not* a property that would be associated with the datatype `Foo *` by itself. Therefore, a completely different SWIG customization mechanism (`%feature`) is used for this purpose. Consult the Customization Features chapter for more information about that.

Typemaps also can't be used to rearrange or transform the order of arguments. For example, if you had a function like this:

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

you can't use typemaps to interchange the arguments, allowing you to call the function like this:

```
foo("hello",3)          # Reversed arguments
```

If you want to change the calling conventions of a function, write a helper function instead. For example:

```
%rename(foo) wrap_foo;
%inline %{
void wrap_foo(char *s, int x) {
   foo(x,s);
}
%}
```

10.1.7 The rest of this chapter

The rest of this chapter provides detailed information for people who want to write new typemaps. This information is of particular importance to anyone who intends to write a new SWIG target language module. Power users can also use this information to write application specific type conversion rules.

Since typemaps are strongly tied to the underlying C++ type system, subsequent sections assume that you are reasonably familiar with the basic details of values, pointers, references, arrays, type qualifiers (e.g., `const`), structures, namespaces, templates, and memory management in C/C++. If not, you would be well-advised to consult a copy of "The C Programming Language" by Kernighan and Ritchie or "The C++ Programming Language" by Stroustrup before going any further.

10.2 Typemap specifications

This section describes the behavior of the `%typemap` directive itself.

10.2.1 Defining a typemap

New typemaps are defined using the `%typemap` declaration. The general form of this declaration is as follows (parts enclosed in [...] are optional):

```
%typemap(method [, modifiers]) typelist code ;
```

method is a simply a name that specifies what kind of typemap is being defined. It is usually a name like `"in"`, `"out"`, or `"argout"`. The purpose of these methods is described later.

modifiers is an optional comma separated list of `name="value"` values. These are sometimes to attach extra information to a typemap and is often target-language dependent.

typelist is a list of the C++ type patterns that the typemap will match. The general form of this list is as follows:

```
typelist    :  typepattern [, typepattern, typepattern, ... ] ;

typepattern :  type [ (parms) ]
            |  type name [ (parms) ]
            |  ( typelist ) [ (parms) ]
```

Each type pattern is either a simple type, a simple type and argument name, or a list of types in the case of multi-argument typemaps. In addition, each type pattern can be parameterized with a list of temporary variables (parms). The purpose of these variables will be explained shortly.

code specifies the code used in the typemap. Usually this is C/C++ code, but in the statically typed target languages, such as Java and C#, this can contain target language code for certain typemaps. It can take any one of the following forms:

```
code        : { ... }
            | " ... "
            | %{ ... %}
```

Note that the preprocessor will expand code within the {} delimiters, but not in the last two styles of delimiters, see Preprocessor and Typemaps. Here are some examples of valid typemap specifications:

```
/* Simple typemap declarations */
%typemap(in) int {
   $1 = PyInt_AsLong($input);
}
%typemap(in) int "$1 = PyInt_AsLong($input);";
```

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

/* Typemap with extra argument name */
%typemap(in) int nonnegative {
   ...
}

/* Multiple types in one typemap */
%typemap(in) int, short, long {
   $1 = SvIV($input);
}

/* Typemap with modifiers */
%typemap(in,doc="integer") int "$1 = gh_scm2int($input);";

/* Typemap applied to patterns of multiple arguments */
%typemap(in) (char *str, int len),
             (char *buffer, int size)
{
   $1 = PyString_AsString($input);
   $2 = PyString_Size($input);
}

/* Typemap with extra pattern parameters */
%typemap(in, numinputs=0) int *output (int temp),
                          long *output (long temp)
{
   $1 = &temp;
}
```

Admittedly, it's not the most readable syntax at first glance. However, the purpose of the individual pieces will become clear.

10.2.2 Typemap scope

Once defined, a typemap remains in effect for all of the declarations that follow. A typemap may be redefined for different sections of an input file. For example:

```
// typemap1
%typemap(in) int {
...
}

int fact(int);                    // typemap1
int gcd(int x, int y);            // typemap1

// typemap2
%typemap(in) int {
...
}

int isprime(int);                 // typemap2
```

One exception to the typemap scoping rules pertains to the `%extend` declaration. `%extend` is used to attach new declarations to a class or structure definition. Because of this, all of the declarations in an `%extend` block are subject to the typemap rules that are in effect at the point where the class itself is defined. For example:

```
class Foo {
   ...
};

%typemap(in) int {
```

```
 ...
}

%extend Foo {
   int blah(int x);    // typemap has no effect.  Declaration is attached to Foo which
                       // appears before the %typemap declaration.
};
```

10.2.3 Copying a typemap

A typemap is copied by using assignment. For example:

```
%typemap(in) Integer = int;
```

or this:

```
%typemap(in) Integer, Number, int32_t = int;
```

Types are often managed by a collection of different typemaps. For example:

```
%typemap(in)     int { ... }
%typemap(out)    int { ... }
%typemap(varin)  int { ... }
%typemap(varout) int { ... }
```

To copy all of these typemaps to a new type, use `%apply`. For example:

```
%apply int { Integer };            // Copy all int typemaps to Integer
%apply int { Integer, Number };    // Copy all int typemaps to both Integer and Number
```

The patterns for `%apply` follow the same rules as for `%typemap`. For example:

```
%apply int *output { Integer *output };                    // Typemap with name
%apply (char *buf, int len) { (char *buffer, int size) };  // Multiple arguments
```

10.2.4 Deleting a typemap

A typemap can be deleted by simply defining no code. For example:

```
%typemap(in) int;               // Clears typemap for int
%typemap(in) int, long, short;  // Clears typemap for int, long, short
%typemap(in) int *output;
```

The `%clear` directive clears all typemaps for a given type. For example:

```
%clear int;                     // Removes all types for int
%clear int *output, long *output;
```

Note: Since SWIG's default behavior is defined by typemaps, clearing a fundamental type like `int` will make that type unusable unless you also define a new set of typemaps immediately after the clear operation.

10.2.5 Placement of typemaps

Typemap declarations can be declared in the global scope, within a C++ namespace, and within a C++ class. For example:

```
%typemap(in) int {
   ...
}

namespace std {
    class string;
    %typemap(in) string {
        ...
    }
}

class Bar {
public:
    typedef const int & const_reference;
    %typemap(out) const_reference {
         ...
    }
};
```

When a typemap appears inside a namespace or class, it stays in effect until the end of the SWIG input (just like before). However, the typemap takes the local scope into account. Therefore, this code

```
namespace std {
    class string;
    %typemap(in) string {
       ...
    }
}
```

is really defining a typemap for the type `std::string`. You could have code like this:

```
namespace std {
    class string;
    %typemap(in) string {          /* std::string */
       ...
    }
}

namespace Foo {
    class string;
    %typemap(in) string {          /* Foo::string */
       ...
    }
}
```

In this case, there are two completely distinct typemaps that apply to two completely different types (`std::string` and `Foo::string`).

It should be noted that for scoping to work, SWIG has to know that `string` is a typename defined within a particular namespace. In this example, this is done using the class declaration `class string`.

10.3 Pattern matching rules

The section describes the pattern matching rules by which C datatypes are associated with typemaps.

10.3.1 Basic matching rules

Typemaps are matched using both a type and a name (typically the name of a argument). For a given `TYPE NAME` pair, the following rules are applied, in order, to find a match. The first typemap found is used.

- Typemaps that exactly match `TYPE` and `NAME`.
- Typemaps that exactly match `TYPE` only.

If `TYPE` includes qualifiers (const, volatile, etc.), they are stripped and the following checks are made:

- Typemaps that match the stripped `TYPE` and `NAME`.
- Typemaps that match the stripped `TYPE` only.

If `TYPE` is an array. The following transformation is made:

- Replace all dimensions to `[ANY]` and look for a generic array typemap.

To illustrate, suppose that you had a function like this:

```
int foo(const char *s);
```

To find a typemap for the argument `const char *s`, SWIG will search for the following typemaps:

```
const char *s           Exact type and name match
const char *            Exact type match
char *s                 Type and name match (stripped qualifiers)
char *                  Type match (stripped qualifiers)
```

When more than one typemap rule might be defined, only the first match found is actually used. Here is an example that shows how some of the basic rules are applied:

```
%typemap(in) int *x {
   ... typemap 1
}

%typemap(in) int * {
   ... typemap 2
}

%typemap(in) const int *z {
   ... typemap 3
}

%typemap(in) int [4] {
   ... typemap 4
}

%typemap(in) int [ANY] {
   ... typemap 5
}

void A(int *x);        // int *x rule    (typemap 1)
void B(int *y);        // int * rule     (typemap 2)
void C(const int *x);  // int *x rule    (typemap 1)
void D(const int *z);  // int * rule     (typemap 3)
void E(int x[4]);      // int [4] rule   (typemap 4)
```

```
void F(int x[1000]);   // int [ANY] rule (typemap 5)
```

10.3.2 Typedef reductions

If no match is found using the rules in the previous section, SWIG applies a typedef reduction to the type and repeats the typemap search for the reduced type. To illustrate, suppose you had code like this:

```
%typemap(in) int {
   ... typemap 1
}

typedef int Integer;
void blah(Integer x);
```

To find the typemap for `Integer x`, SWIG will first search for the following typemaps:

```
Integer x
Integer
```

Finding no match, it then applies a reduction `Integer -> int` to the type and repeats the search.

```
int x
int      --> match: typemap 1
```

Even though two types might be the same via typedef, SWIG allows typemaps to be defined for each typename independently. This allows for interesting customization possibilities based solely on the typename itself. For example, you could write code like this:

```
typedef double  pdouble;     // Positive double

// typemap 1
%typemap(in) double {
   ... get a double ...
}
// typemap 2
%typemap(in) pdouble {
   ... get a positive double ...
}
double sin(double x);           // typemap 1
pdouble sqrt(pdouble x);        // typemap 2
```

When reducing the type, only one typedef reduction is applied at a time. The search process continues to apply reductions until a match is found or until no more reductions can be made.

For complicated types, the reduction process can generate a long list of patterns. Consider the following:

```
typedef int Integer;
typedef Integer Row4[4];
void foo(Row4 rows[10]);
```

To find a match for the `Row4 rows[10]` argument, SWIG would check the following patterns, stopping only when it found a match:

```
Row4 rows[10]
Row4 [10]
Row4 rows[ANY]
Row4 [ANY]

# Reduce Row4 --> Integer[4]
Integer rows[10][4]
Integer [10][4]
Integer rows[ANY][ANY]
Integer [ANY][ANY]

# Reduce Integer --> int
int rows[10][4]
int [10][4]
int rows[ANY][ANY]
int [ANY][ANY]
```

For parameterized types like templates, the situation is even more complicated. Suppose you had some declarations like this:

```
typedef int Integer;
typedef foo<Integer,Integer> fooii;
void blah(fooii *x);
```

In this case, the following typemap patterns are searched for the argument `fooii *x`:

```
fooii *x
fooii *

# Reduce fooii --> foo<Integer,Integer>
foo<Integer,Integer> *x
foo<Integer,Integer> *

# Reduce Integer -> int
foo<int, Integer> *x
foo<int, Integer> *

# Reduce Integer -> int
foo<int, int> *x
foo<int, int> *
```

Typemap reductions are always applied to the left-most type that appears. Only when no reductions can be made to the left-most type are reductions made to other parts of the type. This behavior means that you could define a typemap for `foo<int,Integer>`, but a typemap for `foo<Integer,int>` would never be matched. Admittedly, this is rather esoteric--there's little practical reason to write a typemap quite like that. Of course, you could rely on this to confuse your coworkers even more.

10.3.3 Default typemaps

Most SWIG language modules use typemaps to define the default behavior of the C primitive types. This is entirely straightforward. For example, a set of typemaps are written like this:

```
%typemap(in) int   "convert an int";
%typemap(in) short "convert a short";
%typemap(in) float "convert a float";
...
```

Since typemap matching follows all `typedef` declarations, any sort of type that is mapped to a primitive type through `typedef` will be picked up by one of these primitive typemaps.

The default behavior for pointers, arrays, references, and other kinds of types are handled by specifying rules for variations of the reserved `SWIGTYPE` type. For example:

```
%typemap(in) SWIGTYPE *            { ... default pointer handling ...          }
%typemap(in) SWIGTYPE &            { ... default reference handling ...        }
%typemap(in) SWIGTYPE []           { ... default array handling ...            }
%typemap(in) enum SWIGTYPE         { ... default handling for enum values ... }
%typemap(in) SWIGTYPE (CLASS::*)   { ... default pointer member handling ...   }
```

These rules match any kind of pointer, reference, or array--even when multiple levels of indirection or multiple array dimensions are used. Therefore, if you wanted to change SWIG's default handling for all types of pointers, you would simply redefine the rule for `SWIGTYPE *`.

Finally, the following typemap rule is used to match against simple types that don't match any other rules:

```
%typemap(in) SWIGTYPE   { ... handle an unknown type ... }
```

This typemap is important because it is the rule that gets triggered when call or return by value is used. For instance, if you have a declaration like this:

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

The `Vector` type will usually just get matched against `SWIGTYPE`. The default implementation of `SWIGTYPE` is to convert the value into pointers (as described in chapter 3).

By redefining `SWIGTYPE` it may be possible to implement other behavior. For example, if you cleared all typemaps for `SWIGTYPE`, SWIG simply won't wrap any unknown datatype (which might be useful for debugging). Alternatively, you might modify SWIGTYPE to marshal objects into strings instead of converting them to pointers.

The best way to explore the default typemaps is to look at the ones already defined for a particular language module. Typemaps definitions are usually found in the SWIG library in a file such as `python.swg`, `tcl8.swg`, etc.

10.3.4 Mixed default typemaps

The default typemaps described above can be mixed with `const` and with each other. For example the `SWIGTYPE *` typemap is for default pointer handling, but if a `const SWIGTYPE *` typemap is defined it will be used instead for constant pointers. Some further examples follow:

```
%typemap(in) enum SWIGTYPE &        { ... enum references ...                          }
%typemap(in) const enum SWIGTYPE &  { ... const enum references ...                    }
%typemap(in) SWIGTYPE *&            { ... pointers passed by reference ...             }
%typemap(in) SWIGTYPE * const &     { ... constant pointers passed by reference ... }
%typemap(in) SWIGTYPE[ANY][ANY]     { ... 2D arrays ...                                }
```

Note that the the typedef reduction described earlier is also used with these mixed default typemaps. For example, say the following typemaps are defined and SWIG is looking for the best match for the enum shown below:

```
%typemap(in) const Hello &          { ... }
%typemap(in) const enum SWIGTYPE &  { ... }
%typemap(in) enum SWIGTYPE &        { ... }
%typemap(in) SWIGTYPE &             { ... }
%typemap(in) SWIGTYPE               { ... }
```

```
enum Hello {};
const Hello &hi;
```

The typemap at the top of the list will be chosen, not because it is defined first, but because it is the closest match for the type being wrapped. If any of the typemaps in the above list were not defined, then the next one on the list would have precedence. In other words the typemap chosen is the closest explicit match.

Compatibility note: The mixed default typemaps were introduced in SWIG-1.3.23, but were not used much in this version. Expect to see them being used more and more within the various libraries in later versions of SWIG.

10.3.5 Multi-arguments typemaps

When multi-argument typemaps are specified, they take precedence over any typemaps specified for a single type. For example:

```
%typemap(in) (char *buffer, int len) {
   // typemap 1
}

%typemap(in) char *buffer {
   // typemap 2
}

void foo(char *buffer, int len, int count); // (char *buffer, int len)
void bar(char *buffer, int blah);           // char *buffer
```

Multi-argument typemaps are also more restrictive in the way that they are matched. Currently, the first argument follows the matching rules described in the previous section, but all subsequent arguments must match exactly.

10.4 Code generation rules

This section describes rules by which typemap code is inserted into the generated wrapper code.

10.4.1 Scope

When a typemap is defined like this:

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

the typemap code is inserted into the wrapper function using a new block scope. In other words, the wrapper code will look like this:

```
wrap_whatever() {
    ...
    // Typemap code
    {
       arg1 = PyInt_AsLong(obj1);
    }
    ...
}
```

Because the typemap code is enclosed in its own block, it is legal to declare temporary variables for use during typemap execution. For example:

```
%typemap(in) short {
   long temp;           /* Temporary value */
   if (Tcl_GetLongFromObj(interp, $input, &temp) != TCL_OK) {
      return TCL_ERROR;
   }
   $1 = (short) temp;
}
```

Of course, any variables that you declare inside a typemap are destroyed as soon as the typemap code has executed (they are not visible to other parts of the wrapper function or other typemaps that might use the same variable names).

Occasionally, typemap code will be specified using a few alternative forms. For example:

```
%typemap(in) int "$1 = PyInt_AsLong($input);";
%typemap(in) int %{
$1 = PyInt_AsLong($input);
%}
```

These two forms are mainly used for cosmetics--the specified code is not enclosed inside a block scope when it is emitted. This sometimes results in a less complicated looking wrapper function.

10.4.2 Declaring new local variables

Sometimes it is useful to declare a new local variable that exists within the scope of the entire wrapper function. A good example of this might be an application in which you wanted to marshal strings. Suppose you had a C++ function like this

```
int foo(std::string *s);
```

and you wanted to pass a native string in the target language as an argument. For instance, in Perl, you wanted the function to work like this:

```
$x = foo("Hello World");
```

To do this, you can't just pass a raw Perl string as the `std::string *` argument. Instead, you have to create a temporary `std::string` object, copy the Perl string data into it, and then pass a pointer to the object. To do this, simply specify the typemap with an extra parameter like this:

```
%typemap(in) std::string * (std::string temp) {
    unsigned int len;
    char        *s;
    s = SvPV($input,len);         /* Extract string data */
    temp.assign(s,len);           /* Assign to temp */
    $1 = &temp;                   /* Set argument to point to temp */
}
```

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

```
wrap_foo() {
   std::string temp;    <--- Declaration of temp goes here
   ...

   /* Typemap code */
   {
      ...
      temp.assign(s,len);
      ...
   }
   ...
}
```

When you set `temp` to a value, it persists for the duration of the wrapper function and gets cleaned up automatically on exit.

It is perfectly safe to use more than one typemap involving local variables in the same declaration. For example, you could declare a function as :

```
void foo(std::string *x, std::string *y, std::string *z);
```

This is safely handled because SWIG actually renames all local variable references by appending an argument number suffix. Therefore, the generated code would actually look like this:

```
wrap_foo() {
   int *arg1;    /* Actual arguments */
   int *arg2;
   int *arg3;
   std::string temp1;    /* Locals declared in the typemap */
   std::string temp2;
   std::string temp3;
   ...
   {
       char *s;
       unsigned int len;
       ...
       temp1.assign(s,len);
       arg1 = *temp1;
   }
   {
       char *s;
       unsigned int len;
       ...
       temp2.assign(s,len);
       arg2 = &temp2;
   }
   {
       char *s;
       unsigned int len;
       ...
       temp3.assign(s,len);
       arg3 = &temp3;
   }
   ...
}
```

Some typemaps do not recognize local variables (or they may simply not apply). At this time, only typemaps that apply to argument conversion support this.

Note:

When declaring a typemap for multiple types, each type must have its own local variable declaration.

```
%typemap(in) const std::string *, std::string * (std::string temp) // NO!
// only std::string * has a local variable
// const std::string * does not (oops)
....

%typemap(in) const std::string * (std::string temp), std::string * (std::string temp) // Correct
....
```

10.4.3 Special variables

Within all typemaps, the following special variables are expanded. This is by no means a complete list as some target languages have additional special variables which are documented in the language specific chapters.

Variable	Meaning
$*n*	A C local variable corresponding to type *n* in the typemap pattern.
$argnum	Argument number. Only available in typemaps related to argument conversion
$*n*_name	Argument name
$*n*_type	Real C datatype of type *n*.
$*n*_ltype	ltype of type *n*
$*n*_mangle	Mangled form of type *n*. For example `_p_Foo`
$*n*_descriptor	Type descriptor structure for type *n*. For example `SWIGTYPE_p_Foo`. This is primarily used when interacting with the run-time type checker (described later).
$**n*_type	Real C datatype of type *n* with one pointer removed.
$**n*_ltype	ltype of type *n* with one pointer removed.
$**n*_mangle	Mangled form of type *n* with one pointer removed.
$**n*_descriptor	Type descriptor structure for type *n* with one pointer removed.
$&*n*_type	Real C datatype of type *n* with one pointer added.
$&*n*_ltype	ltype of type *n* with one pointer added.
$&*n*_mangle	Mangled form of type *n* with one pointer added.
$&*n*_descriptor	Type descriptor structure for type *n* with one pointer added.
$*n*_basetype	Base typename with all pointers and qualifiers stripped.

Within the table, $*n* refers to a specific type within the typemap specification. For example, if you write this

```
%typemap(in) int *INPUT {

}
```

then $1 refers to `int *INPUT`. If you have a typemap like this,

```
%typemap(in) (int argc, char *argv[]) {
  ...
}
```

then $1 refers to `int argc` and $2 refers to `char *argv[]`.

Substitutions related to types and names always fill in values from the actual code that was

matched. This is useful when a typemap might match multiple C datatype. For example:

```
%typemap(in)  int, short, long {
   $1 = ($1_ltype) PyInt_AsLong($input);
}
```

In this case, `$1_ltype` is replaced with the datatype that is actually matched.

When typemap code is emitted, the C/C++ datatype of the special variables `$1` and `$2` is always an "ltype." An "ltype" is simply a type that can legally appear on the left-hand side of a C assignment operation. Here are a few examples of types and ltypes:

```
type              ltype
------            ----------------
int               int
const int         int
const int *       int *
int [4]           int *
int [4][5]        int (*)[5]
```

In most cases a ltype is simply the C datatype with qualifiers stripped off. In addition, arrays are converted into pointers.

Variables such as `$&1_type` and `$*1_type` are used to safely modify the type by removing or adding pointers. Although not needed in most typemaps, these substitutions are sometimes needed to properly work with typemaps that convert values between pointers and values.

If necessary, type related substitutions can also be used when declaring locals. For example:

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

There is one word of caution about declaring local variables in this manner. If you declare a local variable using a type substitution such as `$1_ltype temp`, it won't work like you expect for arrays and certain kinds of pointers. For example, if you wrote this,

```
%typemap(in) int [10][20] {
   $1_ltype temp;
}
```

then the declaration of `temp` will be expanded as

```
int (*)[20] temp;
```

This is illegal C syntax and won't compile. There is currently no straightforward way to work around this problem in SWIG due to the way that typemap code is expanded and processed. However, one possible workaround is to simply pick an alternative type such as `void *` and use casts to get the correct type when needed. For example:

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

```
}
```

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

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

10.4.4 Special variable macros

Special variable macros are like macro functions in that they take one or more input arguments which are used for the macro expansion. They look like macro/function calls but use the special variable `$` prefix to the macro name. Note that unlike normal macros, the expansion is not done by the preprocessor, it is done during the SWIG parsing/compilation stages. The following special variable macros are available across all language modules.

10.4.4.1 $descriptor(type)

This macro expands into the type descriptor structure for any C/C++ type specified in `type`. It behaves like the `$1_descriptor` special variable described above except that the type to expand is taken from the macro argument rather than inferred from the typemap type. For example, `$descriptor(std::vector<int> *)` will expand into `SWIGTYPE_p_std__vectorT_int_t`. This macro is mostly used in the scripting target languages and is demonstrated later in the Run-time type checker usage section.

10.4.4.2 $typemap(method, typepattern)

This macro uses the pattern matching rules described earlier to lookup and then substitute the special variable macro with the code in the matched typemap. The typemap to search for is specified by the arguments, where `method` is the typemap method name and `typepattern` is a type pattern as per the `%typemap` specification in the Defining a typemap section.

The special variables within the matched typemap are expanded into those for the matched typemap type, not the typemap within which the macro is called. In practice, there is little use for this macro in the scripting target languages. It is mostly used in the target languages that are statically typed as a way to obtain the target language type given the C/C++ type and more commonly only when the C++ type is a template parameter.

The example below is for C# only and uses some typemap method names documented in the C# chapter, but it shows some of the possible syntax variations.

```
%typemap(cstype) unsigned long    "uint"
%typemap(cstype) unsigned long bb "bool"
%typemap(cscode) BarClass %{
  void foo($typemap(cstype, unsigned long aa) var1,
           $typemap(cstype, unsigned long bb) var2,
           $typemap(cstype, (unsigned long bb)) var3,
           $typemap(cstype, unsigned long) var4)
  {
    // do something
  }
%}
```

The result is the following expansion

```
%typemap(cstype) unsigned long    "uint"
%typemap(cstype) unsigned long bb "bool"
%typemap(cscode) BarClass %{
  void foo(uint var1,
           bool var2,
           bool var3,
           uint var4)
  {
    // do something
  }
%}
```

10.5 Common typemap methods

The set of typemaps recognized by a language module may vary. However, the following typemap methods are nearly universal:

10.5.1 "in" typemap

The "in" typemap is used to convert function arguments from the target language to C. For example:

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

The following special variables are available:

```
$input            - Input object holding value to be converted.
$symname          - Name of function/method being wrapped
```

This is probably the most commonly redefined typemap because it can be used to implement customized conversions.

In addition, the "in" typemap allows the number of converted arguments to be specified. The `numinputs` attributes facilitates this. For example:

```
// Ignored argument.
%typemap(in, numinputs=0) int *out (int temp) {
    $1 = &temp;
}
```

At this time, only zero or one arguments may be converted. When `numinputs` is set to 0, the argument is effectively ignored and cannot be supplied from the target language. The argument is still required when making the C/C++ call and the above typemap shows the value used is instead obtained from a locally declared variable called `temp`. Usually `numinputs` is not specified, whereupon the default value is 1, that is, there is a one to one mapping of the number of arguments when used from the target language to the C/C++ call. Multi-argument typemaps provide a similar concept where the number of arguments mapped from the target language to C/C++ can be changed for more tha multiple adjacent C/C++ arguments.

Compatibility note: Specifying `numinputs=0` is the same as the old "ignore" typemap.

10.5.2 "typecheck" typemap

The "typecheck" typemap is used to support overloaded functions and methods. It merely checks an argument to see whether or not it matches a specific type. For example:

```
%typemap(typecheck,precedence=SWIG_TYPECHECK_INTEGER) int {
   $1 = PyInt_Check($input) ? 1 : 0;
}
```

For typechecking, the $1 variable is always a simple integer that is set to 1 or 0 depending on whether or not the input argument is the correct type.

If you define new "in" typemaps *and* your program uses overloaded methods, you should also define a collection of "typecheck" typemaps. More details about this follow in a later section on "Typemaps and Overloading."

10.5.3 "out" typemap

The "out" typemap is used to convert function/method return values from C into the target language. For example:

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

The following special variables are available.

```
$result           - Result object returned to target language.
$symname          - Name of function/method being wrapped
```

The "out" typemap supports an optional attribute flag called "optimal". This is for code optimisation and is detailed in the Optimal code generation when returning by value section.

10.5.4 "arginit" typemap

The "arginit" typemap is used to set the initial value of a function argument--before any conversion has occurred. This is not normally necessary, but might be useful in highly specialized applications. For example:

```
// Set argument to NULL before any conversion occurs
%typemap(arginit) int *data {
   $1 = NULL;
}
```

10.5.5 "default" typemap

The "default" typemap is used to turn an argument into a default argument. For example:

```
%typemap(default) int flags {
   $1 = DEFAULT_FLAGS;
}
...
int foo(int x, int y, int flags);
```

The primary use of this typemap is to either change the wrapping of default arguments or specify a default argument in a language where they aren't supported (like C). Target languages that do not support optional arguments, such as Java and C#, effectively ignore the

value specified by this typemap as all arguments must be given.

Once a default typemap has been applied to an argument, all arguments that follow must have default values. See the Default/optional arguments section for further information on default argument wrapping.

10.5.6 "check" typemap

The "check" typemap is used to supply value checking code during argument conversion. The typemap is applied *after* arguments have been converted. For example:

```
%typemap(check) int positive {
   if ($1 <= 0) {
       SWIG_exception(SWIG_ValueError,"Expected positive value.");
   }
}
```

10.5.7 "argout" typemap

The "argout" typemap is used to return values from arguments. This is most commonly used to write wrappers for C/C++ functions that need to return multiple values. The "argout" typemap is almost always combined with an "in" typemap---possibly to ignore the input value. For example:

```
/* Set the input argument to point to a temporary variable */
%typemap(in, numinputs=0) int *out (int temp) {
   $1 = &temp;
}

%typemap(argout) int *out {
   // Append output value $1 to $result
   ...
}
```

The following special variables are available.

```
$result           - Result object returned to target language.
$input            - The original input object passed.
$symname          - Name of function/method being wrapped
```

The code supplied to the "argout" typemap is always placed after the "out" typemap. If multiple return values are used, the extra return values are often appended to return value of the function.

See the `typemaps.i` library for examples.

10.5.8 "freearg" typemap

The "freearg" typemap is used to cleanup argument data. It is only used when an argument might have allocated resources that need to be cleaned up when the wrapper function exits. The "freearg" typemap usually cleans up argument resources allocated by the "in" typemap. For example:

```
// Get a list of integers
%typemap(in) int *items {
   int nitems = Length($input);
   $1 = (int *) malloc(sizeof(int)*nitems);
}
```

```
// Free the list
%typemap(freearg) int *items {
   free($1);
}
```

The "freearg" typemap inserted at the end of the wrapper function, just before control is returned back to the target language. This code is also placed into a special variable `$cleanup` that may be used in other typemaps whenever a wrapper function needs to abort prematurely.

10.5.9 "newfree" typemap

The "newfree" typemap is used in conjunction with the `%newobject` directive and is used to deallocate memory used by the return result of a function. For example:

```
%typemap(newfree) string * {
   delete $1;
}
%typemap(out) string * {
   $result = PyString_FromString($1->c_str());
}
...

%newobject foo;
...
string *foo();
```

See Object ownership and %newobject for further details.

10.5.10 "memberin" typemap

The "memberin" typemap is used to copy data from *an already converted input value* into a structure member. It is typically used to handle array members and other special cases. For example:

```
%typemap(memberin) int [4] {
   memmove($1, $input, 4*sizeof(int));
}
```

It is rarely necessary to write "memberin" typemaps---SWIG already provides a default implementation for arrays, strings, and other objects.

10.5.11 "varin" typemap

The "varin" typemap is used to convert objects in the target language to C for the purposes of assigning to a C/C++ global variable. This is implementation specific.

10.5.12 "varout" typemap

The "varout" typemap is used to convert a C/C++ object to an object in the target language when reading a C/C++ global variable. This is implementation specific.

10.5.13 "throws" typemap

The "throws" typemap is only used when SWIG parses a C++ method with an exception specification or has the `%catches` feature attached to the method. It provides a default mechanism for handling C++ methods that have declared the exceptions they will throw. The purpose of this typemap is to convert a C++ exception into an error or exception in the target language. It is slightly different to the other typemaps as it is based around the exception type

rather than the type of a parameter or variable. For example:

```
%typemap(throws) const char * %{
  PyErr_SetString(PyExc_RuntimeError, $1);
  SWIG_fail;
%}
void bar() throw (const char *);
```

As can be seen from the generated code below, SWIG generates an exception handler with the catch block comprising the "throws" typemap content.

```
...
try {
    bar();
}
catch(char const *_e) {
    PyErr_SetString(PyExc_RuntimeError, _e);
    SWIG_fail;

}
...
```

Note that if your methods do not have an exception specification yet they do throw exceptions, SWIG cannot know how to deal with them. For a neat way to handle these, see the Exception handling with %exception section.

10.6 Some typemap examples

This section contains a few examples. Consult language module documentation for more examples.

10.6.1 Typemaps for arrays

A common use of typemaps is to provide support for C arrays appearing both as arguments to functions and as structure members.

For example, suppose you had a function like this:

```
void set_vector(int type, float value[4]);
```

If you wanted to handle `float value[4]` as a list of floats, you might write a typemap similar to this:

```
%typemap(in) float value[4] (float temp[4]) {
  int i;
  if (!PySequence_Check($input)) {
    PyErr_SetString(PyExc_ValueError,"Expected a sequence");
    return NULL;
  }
  if (PySequence_Length($input) != 4) {
    PyErr_SetString(PyExc_ValueError,"Size mismatch. Expected 4 elements");
    return NULL;
  }
  for (i = 0; i < 4; i++) {
    PyObject *o = PySequence_GetItem($input,i);
    if (PyNumber_Check(o)) {
      temp[i] = (float) PyFloat_AsDouble(o);
    } else {
```

```
      PyErr_SetString(PyExc_ValueError,"Sequence elements must be numbers");
      return NULL;
    }
  }
  $1 = temp;
}
```

In this example, the variable `temp` allocates a small array on the C stack. The typemap then populates this array and passes it to the underlying C function.

When used from Python, the typemap allows the following type of function call:

```
>>> set_vector(type, [ 1, 2.5, 5, 20 ])
```

If you wanted to generalize the typemap to apply to arrays of all dimensions you might write this:

```
%typemap(in) float value[ANY] (float temp[$1_dim0]) {
  int i;
  if (!PySequence_Check($input)) {
    PyErr_SetString(PyExc_ValueError,"Expected a sequence");
    return NULL;
  }
  if (PySequence_Length($input) != $1_dim0) {
    PyErr_SetString(PyExc_ValueError,"Size mismatch. Expected $1_dim0 elements");
    return NULL;
  }
  for (i = 0; i < $1_dim0; i++) {
    PyObject *o = PySequence_GetItem($input,i);
    if (PyNumber_Check(o)) {
      temp[i] = (float) PyFloat_AsDouble(o);
    } else {
      PyErr_SetString(PyExc_ValueError,"Sequence elements must be numbers");
      return NULL;
    }
  }
  $1 = temp;
}
```

In this example, the special variable `$1_dim0` is expanded with the actual array dimensions. Multidimensional arrays can be matched in a similar manner. For example:

```
%typemap(in) float matrix[ANY][ANY] (float temp[$1_dim0][$1_dim1]) {
   ... convert a 2d array ...
}
```

For large arrays, it may be impractical to allocate storage on the stack using a temporary variable as shown. To work with heap allocated data, the following technique can be used.

```
%typemap(in) float value[ANY] {
  int i;
  if (!PySequence_Check($input)) {
    PyErr_SetString(PyExc_ValueError,"Expected a sequence");
    return NULL;
  }
  if (PySequence_Length($input) != $1_dim0) {
    PyErr_SetString(PyExc_ValueError,"Size mismatch. Expected $1_dim0 elements");
    return NULL;
  }
  $1 = (float *) malloc($1_dim0*sizeof(float));
  for (i = 0; i < $1_dim0; i++) {
```

```
    PyObject *o = PySequence_GetItem($input,i);
    if (PyNumber_Check(o)) {
      $1[i] = (float) PyFloat_AsDouble(o);
    } else {
      PyErr_SetString(PyExc_ValueError,"Sequence elements must be numbers");
      free($1);
      return NULL;
    }
  }
}
%typemap(freearg) float value[ANY] {
   if ($1) free($1);
}
```

In this case, an array is allocated using `malloc`. The `freearg` typemap is then used to release the argument after the function has been called.

Another common use of array typemaps is to provide support for array structure members. Due to subtle differences between pointers and arrays in C, you can't just "assign" to a array structure member. Instead, you have to explicitly copy elements into the array. For example, suppose you had a structure like this:

```
struct SomeObject {
        float  value[4];
        ...
};
```

When SWIG runs, it won't produce any code to set the `vec` member. You may even get a warning message like this:

```
swig -python  example.i
Generating wrappers for Python
example.i:10.  Warning. Array member value will be read-only.
```

These warning messages indicate that SWIG does not know how you want to set the `vec` field.

To fix this, you can supply a special "memberin" typemap like this:

```
%typemap(memberin) float [ANY] {
  int i;
  for (i = 0; i < $1_dim0; i++) {
      $1[i] = $input[i];
  }
}
```

The memberin typemap is used to set a structure member from data that has already been converted from the target language to C. In this case, `$input` is the local variable in which converted input data is stored. This typemap then copies this data into the structure.

When combined with the earlier typemaps for arrays, the combination of the "in" and "memberin" typemap allows the following usage:

```
>>> s = SomeObject()
>>> s.x = [1, 2.5, 5, 10]
```

Related to structure member input, it may be desirable to return structure members as a new kind of object. For example, in this example, you will get very odd program behavior where the structure member can be set nicely, but reading the member simply returns a pointer:

```
>>> s = SomeObject()
>>> s.x = [1, 2.5, 5, 10]
>>> print s.x
_1008fea8_p_float
>>>
```

To fix this, you can write an "out" typemap. For example:

```
%typemap(out) float [ANY] {
  int i;
  $result = PyList_New($1_dim0);
  for (i = 0; i < $1_dim0; i++) {
    PyObject *o = PyFloat_FromDouble((double) $1[i]);
    PyList_SetItem($result,i,o);
  }
}
```

Now, you will find that member access is quite nice:

```
>>> s = SomeObject()
>>> s.x = [1, 2.5, 5, 10]
>>> print s.x
[ 1, 2.5, 5, 10]
```

Compatibility Note: SWIG1.1 used to provide a special "memberout" typemap. However, it was mostly useless and has since been eliminated. To return structure members, simply use the "out" typemap.

10.6.2 Implementing constraints with typemaps

One particularly interesting application of typemaps is the implementation of argument constraints. This can be done with the "check" typemap. When used, this allows you to provide code for checking the values of function arguments. For example :

```
%module math

%typemap(check) double posdouble {
        if ($1 < 0) {
                croak("Expecting a positive number");
        }
}

...
double sqrt(double posdouble);
```

This provides a sanity check to your wrapper function. If a negative number is passed to this function, a Perl exception will be raised and your program terminated with an error message.

This kind of checking can be particularly useful when working with pointers. For example :

```
%typemap(check) Vector * {
    if ($1 == 0) {
        PyErr_SetString(PyExc_TypeError,"NULL Pointer not allowed");
        return NULL;
   }
}
```

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

Note: A more advanced constraint checking system is in development. Stay tuned.

10.7 Typemaps for multiple languages

The code within typemaps is usually language dependent, however, many languages support the same typemaps. In order to distinguish typemaps across different languages, the preprocessor should be used. For example, the "in" typemap for Perl and Ruby could be written as:

```
#if defined(SWIGPERL)
  %typemap(in) int "$1 = ($1_ltype) SvIV($input);"
#elif defined(SWIGRUBY)
  %typemap(in) int "$1 = NUM2INT($input);"
#else
  #warning no "in" typemap defined
#endif
```

The full set of language specific macros is defined in the Conditional Compilation section. The example above also shows a common approach of issuing a warning for an as yet unsupported language.

Compatibility note: In SWIG-1.1 different languages could be distinguished with the language name being put within the `%typemap` directive, for example, `%typemap(ruby,in) int "$1 = NUM2INT($input);".`

10.8 Optimal code generation when returning by value

The "out" typemap is the main typemap for return types. This typemap supports an optional attribute flag called "optimal", which is for reducing temporary variables and the amount of generated code, thereby giving the compiler the opportunity to use *return value optimization* for generating faster executing code. It only really makes a difference when returning objects by value and has some limitations on usage, as explained later on.

When a function returns an object by value, SWIG generates code that instantiates the default type on the stack then assigns the value returned by the function call to it. A copy of this object is then made on the heap and this is what is ultimately stored and used from the target language. This will be clearer considering an example. Consider running the following code through SWIG:

```
%typemap(out) SWIGTYPE %{
  $result = new $1_ltype((const $1_ltype &)$1);
%}

%inline %{
#include <iostream>
using namespace std;

struct XX {
  XX() { cout << "XX()" << endl; }
  XX(int i) { cout << "XX(" << i << ")" << endl; }
  XX(const XX &other) { cout << "XX(const XX &)" << endl; }
  XX & operator =(const XX &other) { cout << "operator=(const XX &)" << endl; return *this; }
  ~XX() { cout << "~XX()" << endl; }
  static XX create() {
    return XX(0);
```

```
  }
};
%}
```

The "out" typemap shown is the default typemap for C# when returning by objects by value. When making a call to `XX::create()` from C#, the output is as follows:

```
XX()
XX(0)
operator=(const XX &)
~XX()
XX(const XX &)
~XX()
~XX()
```

Note that three objects are being created as well as an assignment. Wouldn't it be great if the `XX::create()` method was the only time a constructor was called? As the method returns by value, this is asking a lot and the code that SWIG generates by default makes it impossible for the compiler to use *return value optimisation (RVO)*. However, this is where the "optimal" attribute in the "out" typemap can help out. If the typemap code is kept the same and just the "optimal" attribute specified like this:

```
%typemap(out, optimal="1") SWIGTYPE %{
  $result = new $1_ltype((const $1_ltype &)$1);
%}
```

then when the code is run again, the output is simply:

```
XX(0)
~XX()
```

How the "optimal" attribute works is best explained using the generated code. Without "optimal", the generated code is:

```
SWIGEXPORT void * SWIGSTDCALL CSharp_XX_create() {
  void * jresult ;
  XX result;
  result = XX::create();
  jresult = new XX((const XX &)result);
  return jresult;
}
```

With the "optimal" attribute, the code is:

```
SWIGEXPORT void * SWIGSTDCALL CSharp_XX_create() {
  void * jresult ;
  jresult = new XX((const XX &)XX::create());
  return jresult;
}
```

The major difference is the `result` temporary variable holding the value returned from `XX::create()` is no longer generated and instead the copy constructor call is made directly from the value returned by `XX::create()`. With modern compilers implementing RVO, the copy is not actually done, in fact the object is never created on the stack in `XX::create()` at all, it is simply created directly on the heap. In the first instance, the `$1` special variable in the

typemap is expanded into `result`. In the second instance, `$1` is expanded into `XX::create()` and this is essentially what the "optimal" attribute is telling SWIG to do.

The "optimal" attribute optimisation is not turned on by default as it has a number of restrictions. Firstly, some code cannot be condensed into a simple call for passing into the copy constructor. One common occurrence is when %exception is used. Consider adding the following `%exception` to the example:

```
%exception XX::create() %{
try {
  $action
} catch(const std::exception &e) {
  cout << e.what() << endl;
}
%}
```

SWIG can detect when the "optimal" attribute cannot be used and will ignore it and in this case will issue the following warning:

```
example.i:28: Warning(474): Method XX::create() usage of the optimal attribute in the out
typemap at example.i:14 ignored as the following cannot be used to generate optimal code:
try {
  result = XX::create();
} catch(const std::exception &e) {
  cout << e.what() << endl;
}
```

It should be clear that the above code cannot be used as the argument to the copy constructor call, ie for the `$1` substitution.

Secondly, if the typemaps uses `$1` more than once, then multiple calls to the wrapped function will be made. Obviously that is not very optimal. In fact SWIG attempts to detect this and will issue a warning something like:

```
example.i:21: Warning(475): Multiple calls to XX::create() might be generated due to
optimal attribute usage in the out typemap at example.i:7.
```

However, it doesn't always get it right, for example when `$1` is within some commented out code.

10.9 Multi-argument typemaps

So far, the typemaps presented have focused on the problem of dealing with single values. For example, converting a single input object to a single argument in a function call. However, certain conversion problems are difficult to handle in this manner. As an example, consider the example at the very beginning of this chapter:

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

Suppose that you wanted to wrap this function so that it accepted a single list of strings like this:

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

To do this, you not only need to map a list of strings to `char *argv[]`, but the value of `int argc` is implicitly determined by the length of the list. Using only simple typemaps, this type of conversion is possible, but extremely painful. Therefore, SWIG1.3 introduces the notion of multi-argument typemaps.

A multi-argument typemap is a conversion rule that specifies how to convert a *single* object in the target language to set of consecutive function arguments in C/C++. For example, the following multi-argument maps perform the conversion described for the above example:

```
%typemap(in) (int argc, char *argv[]) {
  int i;
  if (!PyList_Check($input)) {
    PyErr_SetString(PyExc_ValueError, "Expecting a list");
    return NULL;
  }
  $1 = PyList_Size($input);
  $2 = (char **) malloc(($1+1)*sizeof(char *));
  for (i = 0; i < $1; i++) {
    PyObject *s = PyList_GetItem($input,i);
    if (!PyString_Check(s)) {
        free($2);
        PyErr_SetString(PyExc_ValueError, "List items must be strings");
        return NULL;
    }
    $2[i] = PyString_AsString(s);
  }
  $2[i] = 0;
}

%typemap(freearg) (int argc, char *argv[]) {
   if ($2) free($2);
}
```

A multi-argument map is always specified by surrounding the arguments with parentheses as shown. For example:

```
%typemap(in) (int argc, char *argv[]) { ... }
```

Within the typemap code, the variables `$1`, `$2`, and so forth refer to each type in the map. All of the usual substitutions apply--just use the appropriate `$1` or `$2` prefix on the variable name (e.g., `$2_type`, `$1_ltype`, etc.)

Multi-argument typemaps always have precedence over simple typemaps and SWIG always performs longest-match searching. Therefore, you will get the following behavior:

```
%typemap(in) int argc                               { ... typemap 1 ... }
%typemap(in) (int argc, char *argv[])               { ... typemap 2 ... }
%typemap(in) (int argc, char *argv[], char *env[]) { ... typemap 3 ... }

int foo(int argc, char *argv[]);                    // Uses typemap 2
int bar(int argc, int x);                           // Uses typemap 1
int spam(int argc, char *argv[], char *env[]);      // Uses typemap 3
```

It should be stressed that multi-argument typemaps can appear anywhere in a function declaration and can appear more than once. For example, you could write this:

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

void search_words(int scount, char *swords[], int wcount, char *words[], int maxcount);
```

Other directives such as `%apply` and `%clear` also work with multi-argument maps. For example:

```
%apply (int argc, char *argv[]) {
    (int scount, char *swords[]),
    (int wcount, char *words[])
};
...
%clear (int scount, char *swords[]), (int wcount, char *words[]);
...
```

Although multi-argument typemaps may seem like an exotic, little used feature, there are several situations where they make sense. First, suppose you wanted to wrap functions similar to the low-level `read()` and `write()` system calls. For example:

```
typedef unsigned int size_t;

int read(int fd, void *rbuffer, size_t len);
int write(int fd, void *wbuffer, size_t len);
```

As is, the only way to use the functions would be to allocate memory and pass some kind of pointer as the second argument---a process that might require the use of a helper function. However, using multi-argument maps, the functions can be transformed into something more natural. For example, you might write typemaps like this:

```
// typemap for an outgoing buffer
%typemap(in) (void *wbuffer, size_t len) {
   if (!PyString_Check($input)) {
       PyErr_SetString(PyExc_ValueError, "Expecting a string");
       return NULL;
   }
   $1 = (void *) PyString_AsString($input);
   $2 = PyString_Size($input);
}

// typemap for an incoming buffer
%typemap(in) (void *rbuffer, size_t len) {
   if (!PyInt_Check($input)) {
       PyErr_SetString(PyExc_ValueError, "Expecting an integer");
       return NULL;
   }
   $2 = PyInt_AsLong($input);
   if ($2 < 0) {
       PyErr_SetString(PyExc_ValueError, "Positive integer expected");
       return NULL;
   }
   $1 = (void *) malloc($2);
}

// Return the buffer.  Discarding any previous return result
%typemap(argout) (void *rbuffer, size_t len) {
   Py_XDECREF($result);   /* Blow away any previous result */
   if (result < 0) {      /* Check for I/O error */
       free($1);
       PyErr_SetFromErrno(PyExc_IOError);
       return NULL;
   }
   $result = PyString_FromStringAndSize($1,result);
   free($1);
}
```

(note: In the above example, `$result` and `result` are two different variables. `result` is the real C datatype that was returned by the function. `$result` is the scripting language object

being returned to the interpreter.).

Now, in a script, you can write code that simply passes buffers as strings like this:

```
>>> f = example.open("Makefile")
>>> example.read(f,40)
'TOP        = ../..\nSWIG       = $(TOP)/.'
>>> example.read(f,40)
'./swig\nSRCS       = example.c\nTARGET    '
>>> example.close(f)
0
>>> g = example.open("foo", example.O_WRONLY | example.O_CREAT, 0644)
>>> example.write(g,"Hello world\n")
12
>>> example.write(g,"This is a test\n")
15
>>> example.close(g)
0
>>>
```

A number of multi-argument typemap problems also arise in libraries that perform matrix-calculations--especially if they are mapped onto low-level Fortran or C code. For example, you might have a function like this:

```
int is_symmetric(double *mat, int rows, int columns);
```

In this case, you might want to pass some kind of higher-level object as an matrix. To do this, you could write a multi-argument typemap like this:

```
%typemap(in) (double *mat, int rows, int columns) {
    MatrixObject *a;
    a = GetMatrixFromObject($input);     /* Get matrix somehow */

    /* Get matrix properties */
    $1 = GetPointer(a);
    $2 = GetRows(a);
    $3 = GetColumns(a);
}
```

This kind of technique can be used to hook into scripting-language matrix packages such as Numeric Python. However, it should also be stressed that some care is in order. For example, when crossing languages you may need to worry about issues such as row-major vs. column-major ordering (and perform conversions if needed).

10.10 The run-time type checker

Most scripting languages need type information at run-time. This type information can include how to construct types, how to garbage collect types, and the inheritance relationships between types. If the language interface does not provide its own type information storage, the generated SWIG code needs to provide it.

Requirements for the type system:

- Store inheritance and type equivalence information and be able to correctly re-create the type pointer.
- Share type information between modules.
- Modules can be loaded in any order, irregardless of actual type dependency.
- Avoid the use of dynamically allocated memory, and library/system calls in general.
- Provide a reasonably fast implementation, minimizing the lookup time for all language

modules.
- Custom, language specific information can be attached to types.
- Modules can be unloaded from the type system.

10.10.1 Implementation

The run-time type checker is used by many, but not all, of SWIG's supported target languages. The run-time type checker features are not required and are thus not used for strongly typed languages such as Java and C#. The scripting and scheme based languages rely on it and it forms a critical part of SWIG's operation for these languages.

When pointers, arrays, and objects are wrapped by SWIG, they are normally converted into typed pointer objects. For example, an instance of `Foo *` might be a string encoded like this:

```
_108e688_p_Foo
```

At a basic level, the type checker simply restores some type-safety to extension modules. However, the type checker is also responsible for making sure that wrapped C++ classes are handled correctly---especially when inheritance is used. This is especially important when an extension module makes use of multiple inheritance. For example:

```
class Foo {
   int x;
};

class Bar {
   int y;
};

class FooBar : public Foo, public Bar {
   int z;
};
```

When the class `FooBar` is organized in memory, it contains the contents of the classes `Foo` and `Bar` as well as its own data members. For example:

```
FooBar --> | -----------|  <-- Foo
           |   int x    |
           |------------|  <-- Bar
           |   int y    |
           |------------|
           |   int z    |
           |------------|
```

Because of the way that base class data is stacked together, the casting of a `Foobar *` to either of the base classes may change the actual value of the pointer. This means that it is generally not safe to represent pointers using a simple integer or a bare `void *`---type tags are needed to implement correct handling of pointer values (and to make adjustments when needed).

In the wrapper code generated for each language, pointers are handled through the use of special type descriptors and conversion functions. For example, if you look at the wrapper code for Python, you will see code like this:

```
if ((SWIG_ConvertPtr(obj0,(void **) &arg1, SWIGTYPE_p_Foo,1)) == -1) return NULL;
```

In this code, `SWIGTYPE_p_Foo` is the type descriptor that describes `Foo *`. The type descriptor is actually a pointer to a structure that contains information about the type name to use in the

target language, a list of equivalent typenames (via typedef or inheritance), and pointer value handling information (if applicable). The `SWIG_ConvertPtr()` function is simply a utility function that takes a pointer object in the target language and a type-descriptor objects and uses this information to generate a C++ pointer. However, the exact name and calling conventions of the conversion function depends on the target language (see language specific chapters for details).

The actual type code is in swigrun.swg, and gets inserted near the top of the generated swig wrapper file. The phrase "a type X that can cast into a type Y" means that given a type X, it can be converted into a type Y. In other words, X is a derived class of Y or X is a typedef of Y. The structure to store type information looks like this:

```
/* Structure to store information on one type */
typedef struct swig_type_info {
  const char *name;             /* mangled name of this type */
  const char *str;              /* human readable name for this type */
  swig_dycast_func dcast;       /* dynamic cast function down a hierarchy */
  struct swig_cast_info *cast;  /* Linked list of types that can cast into this type */
  void *clientdata;             /* Language specific type data */
} swig_type_info;

/* Structure to store a type and conversion function used for casting */
typedef struct swig_cast_info {
  swig_type_info *type;         /* pointer to type that is equivalent to this type */
  swig_converter_func converter; /* function to cast the void pointers */
  struct swig_cast_info *next;   /* pointer to next cast in linked list */
  struct swig_cast_info *prev;   /* pointer to the previous cast */
} swig_cast_info;
```

Each `swig_type_info` stores a linked list of types that it is equivalent to. Each entry in this doubly linked list stores a pointer back to another swig_type_info structure, along with a pointer to a conversion function. This conversion function is used to solve the above problem of the FooBar class, correctly returning a pointer to the type we want.

The basic problem we need to solve is verifying and building arguments passed to functions. So going back to the `SWIG_ConvertPtr()` function example from above, we are expecting a `Foo *` and need to check if `obj0` is in fact a `Foo *`. From before, `SWIGTYPE_p_Foo` is just a pointer to the `swig_type_info` structure describing `Foo *`. So we loop through the linked list of `swig_cast_info` structures attached to `SWIGTYPE_p_Foo`. If we see that the type of `obj0` is in the linked list, we pass the object through the associated conversion function and then return a positive. If we reach the end of the linked list without a match, then `obj0` can not be converted to a `Foo *` and an error is generated.

Another issue needing to be addressed is sharing type information between multiple modules. More explicitly, we need to have ONE `swig_type_info` for each type. If two modules both use the type, the second module loaded must lookup and use the swig_type_info structure from the module already loaded. Because no dynamic memory is used and the circular dependencies of the casting information, loading the type information is somewhat tricky, and not explained here. A complete description is in the `Lib/swiginit.swg` file (and near the top of any generated file).

Each module has one swig_module_info structure which looks like this:

```
/* Structure used to store module information
 * Each module generates one structure like this, and the runtime collects
 * all of these structures and stores them in a circularly linked list.*/
typedef struct swig_module_info {
  swig_type_info **types;         /* Array of pointers to swig_type_info structs in this module */
  int size;                       /* Number of types in this module */
  struct swig_module_info *next;  /* Pointer to next element in circularly linked list */
  swig_type_info **type_initial;  /* Array of initially generated type structures */
  swig_cast_info **cast_initial;  /* Array of initially generated casting structures */
  void *clientdata;               /* Language specific module data */
```

```
} swig_module_info;
```

Each module stores an array of pointers to `swig_type_info` structures and the number of types in this module. So when a second module is loaded, it finds the `swig_module_info` structure for the first module and searches the array of types. If any of its own types are in the first module and have already been loaded, it uses those `swig_type_info` structures rather than creating new ones. These `swig_module_info` structures are chained together in a circularly linked list.

10.10.2 Usage

This section covers how to use these functions from typemaps. To learn how to call these functions from external files (not the generated _wrap.c file), see the External access to the run-time system section.

When pointers are converted in a typemap, the typemap code often looks similar to this:

```
%typemap(in) Foo * {
  if ((SWIG_ConvertPtr($input, (void **) &$1, $1_descriptor)) == -1) return NULL;
}
```

The most critical part is the typemap is the use of the `$1_descriptor` special variable. When placed in a typemap, this is expanded into the `SWIGTYPE_*` type descriptor object above. As a general rule, you should always use `$1_descriptor` instead of trying to hard-code the type descriptor name directly.

There is another reason why you should always use the `$1_descriptor` variable. When this special variable is expanded, SWIG marks the corresponding type as "in use." When type-tables and type information is emitted in the wrapper file, descriptor information is only generated for those datatypes that were actually used in the interface. This greatly reduces the size of the type tables and improves efficiency.

Occasionally, you might need to write a typemap that needs to convert pointers of other types. To handle this, the special variable macro `$descriptor(type)` covered earlier can be used to generate the SWIG type descriptor name for any C datatype. For example:

```
%typemap(in) Foo * {
  if ((SWIG_ConvertPtr($input, (void **) &$1, $1_descriptor)) == -1) {
     Bar *temp;
     if ((SWIG_ConvertPtr($input, (void **) &temp, $descriptor(Bar *)) == -1) {
         return NULL;
     }
     $1 = (Foo *) temp;
  }
}
```

The primary use of `$descriptor(type)` is when writing typemaps for container objects and other complex data structures. There are some restrictions on the argument---namely it must be a fully defined C datatype. It can not be any of the special typemap variables.

In certain cases, SWIG may not generate type-descriptors like you expect. For example, if you are converting pointers in some non-standard way or working with an unusual combination of interface files and modules, you may find that SWIG omits information for a specific type descriptor. To fix this, you may need to use the `%types` directive. For example:

```
%types(int *, short *, long *, float *, double *);
```

When `%types` is used, SWIG generates type-descriptor information even if those datatypes never appear elsewhere in the interface file.

Further details about the run-time type checking can be found in the documentation for individual language modules. Reading the source code may also help. The file `Lib/swigrun.swg` in the SWIG library contains all of the source code for type-checking. This code is also included in every generated wrapped file so you probably just look at the output of SWIG to get a better sense for how types are managed.

10.11 Typemaps and overloading

In many target languages, SWIG fully supports C++ overloaded methods and functions. For example, if you have a collection of functions like this:

```
int foo(int x);
int foo(double x);
int foo(char *s, int y);
```

You can access the functions in a normal way from the scripting interpreter:

```
# Python
foo(3)           # foo(int)
foo(3.5)         # foo(double)
foo("hello",5)   # foo(char *, int)

# Tcl
foo 3            # foo(int)
foo 3.5          # foo(double)
foo hello 5      # foo(char *, int)
```

To implement overloading, SWIG generates a separate wrapper function for each overloaded method. For example, the above functions would produce something roughly like this:

```
// wrapper pseudocode
_wrap_foo_0(argc, args[]) {       // foo(int)
   int arg1;
   int result;
   ...
   arg1 = FromInteger(args[0]);
   result = foo(arg1);
   return ToInteger(result);
}

_wrap_foo_1(argc, args[]) {       // foo(double)
   double arg1;
   int result;
   ...
   arg1 = FromDouble(args[0]);
   result = foo(arg1);
   return ToInteger(result);
}

_wrap_foo_2(argc, args[]) {       // foo(char *, int)
   char *arg1;
   int   arg2;
   int result;
   ...
   arg1 = FromString(args[0]);
   arg2 = FromInteger(args[1]);
   result = foo(arg1,arg2);
   return ToInteger(result);
}
```

Next, a dynamic dispatch function is generated:

```
_wrap_foo(argc, args[]) {
   if (argc == 1) {
       if (IsInteger(args[0])) {
           return _wrap_foo_0(argc,args);
       }
       if (IsDouble(args[0])) {
           return _wrap_foo_1(argc,args);
       }
   }
   if (argc == 2) {
       if (IsString(args[0]) && IsInteger(args[1])) {
          return _wrap_foo_2(argc,args);
       }
   }
   error("No matching function!\n");
}
```

The purpose of the dynamic dispatch function is to select the appropriate C++ function based on argument types---a task that must be performed at runtime in most of SWIG's target languages.

The generation of the dynamic dispatch function is a relatively tricky affair. Not only must input typemaps be taken into account (these typemaps can radically change the types of arguments accepted), but overloaded methods must also be sorted and checked in a very specific order to resolve potential ambiguity. A high-level overview of this ranking process is found in the "SWIG and C++" chapter. What isn't mentioned in that chapter is the mechanism by which it is implemented---as a collection of typemaps.

To support dynamic dispatch, SWIG first defines a general purpose type hierarchy as follows:

```
Symbolic Name                  Precedence Value
-----------------------------  ----------------
SWIG_TYPECHECK_POINTER          0
SWIG_TYPECHECK_VOIDPTR          10
SWIG_TYPECHECK_BOOL             15
SWIG_TYPECHECK_UINT8            20
SWIG_TYPECHECK_INT8             25
SWIG_TYPECHECK_UINT16           30
SWIG_TYPECHECK_INT16            35
SWIG_TYPECHECK_UINT32           40
SWIG_TYPECHECK_INT32            45
SWIG_TYPECHECK_UINT64           50
SWIG_TYPECHECK_INT64            55
SWIG_TYPECHECK_UINT128          60
SWIG_TYPECHECK_INT128           65
SWIG_TYPECHECK_INTEGER          70
SWIG_TYPECHECK_FLOAT            80
SWIG_TYPECHECK_DOUBLE           90
SWIG_TYPECHECK_COMPLEX          100
SWIG_TYPECHECK_UNICHAR          110
SWIG_TYPECHECK_UNISTRING        120
SWIG_TYPECHECK_CHAR             130
SWIG_TYPECHECK_STRING           140
SWIG_TYPECHECK_BOOL_ARRAY       1015
SWIG_TYPECHECK_INT8_ARRAY       1025
SWIG_TYPECHECK_INT16_ARRAY      1035
SWIG_TYPECHECK_INT32_ARRAY      1045
SWIG_TYPECHECK_INT64_ARRAY      1055
SWIG_TYPECHECK_INT128_ARRAY     1065
SWIG_TYPECHECK_FLOAT_ARRAY      1080
SWIG_TYPECHECK_DOUBLE_ARRAY     1090
```

```
SWIG_TYPECHECK_CHAR_ARRAY        1130
SWIG_TYPECHECK_STRING_ARRAY      1140
```

(These precedence levels are defined in `swig.swg`, a library file that's included by all target language modules.)

In this table, the precedence-level determines the order in which types are going to be checked. Low values are always checked before higher values. For example, integers are checked before floats, single values are checked before arrays, and so forth.

Using the above table as a guide, each target language defines a collection of "typecheck" typemaps. The follow excerpt from the Python module illustrates this:

```
/* Python type checking rules */
/* Note:  %typecheck(X) is a macro for %typemap(typecheck,precedence=X) */

%typecheck(SWIG_TYPECHECK_INTEGER)
         int, short, long,
         unsigned int, unsigned short, unsigned long,
         signed char, unsigned char,
         long long, unsigned long long,
         const int &, const short &, const long &,
         const unsigned int &, const unsigned short &, const unsigned long &,
         const long long &, const unsigned long long &,
         enum SWIGTYPE,
         bool, const bool &
{
  $1 = (PyInt_Check($input) || PyLong_Check($input)) ? 1 : 0;
}

%typecheck(SWIG_TYPECHECK_DOUBLE)
        float, double,
        const float &, const double &
{
  $1 = (PyFloat_Check($input) || PyInt_Check($input) || PyLong_Check($input)) ? 1 : 0;
}

%typecheck(SWIG_TYPECHECK_CHAR) char {
  $1 = (PyString_Check($input) && (PyString_Size($input) == 1)) ? 1 : 0;
}

%typecheck(SWIG_TYPECHECK_STRING) char * {
  $1 = PyString_Check($input) ? 1 : 0;
}

%typecheck(SWIG_TYPECHECK_POINTER) SWIGTYPE *, SWIGTYPE &, SWIGTYPE [] {
  void *ptr;
  if (SWIG_ConvertPtr($input, (void **) &ptr, $1_descriptor, 0) == -1) {
    $1 = 0;
    PyErr_Clear();
  } else {
    $1 = 1;
  }
}

%typecheck(SWIG_TYPECHECK_POINTER) SWIGTYPE {
  void *ptr;
  if (SWIG_ConvertPtr($input, (void **) &ptr, $&1_descriptor, 0) == -1) {
    $1 = 0;
    PyErr_Clear();
  } else {
    $1 = 1;
  }
}

%typecheck(SWIG_TYPECHECK_VOIDPTR) void * {
  void *ptr;
  if (SWIG_ConvertPtr($input, (void **) &ptr, 0, 0) == -1) {
```

```
    $1 = 0;
    PyErr_Clear();
  } else {
    $1 = 1;
  }
}

%typecheck(SWIG_TYPECHECK_POINTER) PyObject *
{
  $1 = ($input != 0);
}
```

It might take a bit of contemplation, but this code has merely organized all of the basic C++ types, provided some simple type-checking code, and assigned each type a precedence value.

Finally, to generate the dynamic dispatch function, SWIG uses the following algorithm:

- Overloaded methods are first sorted by the number of required arguments.
- Methods with the same number of arguments are then sorted by precedence values of argument types.
- Typecheck typemaps are then emitted to produce a dispatch function that checks arguments in the correct order.

If you haven't written any typemaps of your own, it is unnecessary to worry about the typechecking rules. However, if you have written new input typemaps, you might have to supply a typechecking rule as well. An easy way to do this is to simply copy one of the existing typechecking rules. Here is an example,

```
// Typemap for a C++ string
%typemap(in) std::string {
    if (PyString_Check($input)) {
         $1 = std::string(PyString_AsString($input));
     } else {
         SWIG_exception(SWIG_TypeError, "string expected");
     }
}
// Copy the typecheck code for "char *".
%typemap(typecheck) std::string = char *;
```

The bottom line: If you are writing new typemaps and you are using overloaded methods, you will probably have to write typecheck code or copy existing code. Since this is a relatively new SWIG feature, there are few examples to work with. However, you might look at some of the existing library files likes 'typemaps.i' for a guide.

Notes:

- Typecheck typemaps are not used for non-overloaded methods. Because of this, it is still always necessary to check types in any "in" typemaps.
- The dynamic dispatch process is only meant to be a heuristic. There are many corner cases where SWIG simply can't disambiguate types to the same degree as C++. The only way to resolve this ambiguity is to use the %rename directive to rename one of the overloaded methods (effectively eliminating overloading).
- Typechecking may be partial. For example, if working with arrays, the typecheck code might simply check the type of the first array element and use that to dispatch to the correct function. Subsequent "in" typemaps would then perform more extensive type-checking.
- Make sure you read the section on overloading in the "SWIG and C++" chapter.

10.12 More about `%apply` and `%clear`

In order to implement certain kinds of program behavior, it is sometimes necessary to write

sets of typemaps. For example, to support output arguments, one often writes a set of typemaps like this:

```
%typemap(in,numinputs=0) int *OUTPUT (int temp) {
   $1 = &temp;
}
%typemap(argout) int *OUTPUT {
   // return value somehow
}
```

To make it easier to apply the typemap to different argument types and names, the `%apply` directive performs a copy of all typemaps from one type to another. For example, if you specify this,

```
%apply int *OUTPUT { int *retvalue, int32 *output };
```

then all of the `int *OUTPUT` typemaps are copied to `int *retvalue` and `int32 *output`.

However, there is a subtle aspect of `%apply` that needs more description. Namely, `%apply` does not overwrite a typemap rule if it is already defined for the target datatype. This behavior allows you to do two things:

- You can specialize parts of a complex typemap rule by first defining a few typemaps and then using `%apply` to incorporate the remaining pieces.
- Sets of different typemaps can be applied to the same datatype using repeated `%apply` directives.

For example:

```
%typemap(in) int *INPUT (int temp) {
   temp = ... get value from $input ...;
   $1 = &temp;
}

%typemap(check) int *POSITIVE {
   if (*$1 <= 0) {
      SWIG_exception(SWIG_ValueError,"Expected a positive number!\n");
      return NULL;
   }
}

...
%apply int *INPUT     { int *invalue };
%apply int *POSITIVE  { int *invalue };
```

Since `%apply` does not overwrite or replace any existing rules, the only way to reset behavior is to use the `%clear` directive. `%clear` removes all typemap rules defined for a specific datatype. For example:

```
%clear int *invalue;
```

10.13 Reducing wrapper code size

Since the code supplied to a typemap is inlined directly into wrapper functions, typemaps can result in a tremendous amount of code bloat. For example, consider this typemap for an array:

```
%typemap(in) float [ANY] {
  int i;
  if (!PySequence_Check($input)) {
    PyErr_SetString(PyExc_ValueError,"Expected a sequence");
    return NULL;
  }
  if (PySequence_Length($input) != $1_dim0) {
    PyErr_SetString(PyExc_ValueError,"Size mismatch. Expected $1_dim0 elements");
    return NULL;
  }
  $1 = (float) malloc($1_dim0*sizeof(float));
  for (i = 0; i < $1_dim0; i++) {
    PyObject *o = PySequence_GetItem($input,i);
    if (PyNumber_Check(o)) {
      $1[i] = (float) PyFloat_AsDouble(o);
    } else {
      PyErr_SetString(PyExc_ValueError,"Sequence elements must be numbers");
      free(result);
      return NULL;
    }
  }
}
```

If you had a large interface with hundreds of functions all accepting array parameters, this typemap would be replicated repeatedly--generating a huge amount of code. A better approach might be to consolidate some of the typemap into a function. For example:

```
%{
/* Define a helper function */
static float *
convert_float_array(PyObject *input, int size) {
  int i;
  float *result;
  if (!PySequence_Check(input)) {
    PyErr_SetString(PyExc_ValueError,"Expected a sequence");
    return NULL;
  }
  if (PySequence_Length(input) != size) {
    PyErr_SetString(PyExc_ValueError,"Size mismatch. ");
    return NULL;
  }
  result = (float) malloc(size*sizeof(float));
  for (i = 0; i < size; i++) {
    PyObject *o = PySequence_GetItem(input,i);
    if (PyNumber_Check(o)) {
      result[i] = (float) PyFloat_AsDouble(o);
    } else {
      PyErr_SetString(PyExc_ValueError,"Sequence elements must be numbers");
      free(result);
      return NULL;
    }
  }
  return result;
}
%}

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

10.14 Passing data between typemaps

It is also important to note that the primary use of local variables is to create stack-allocated objects for temporary use inside a wrapper function (this is faster and less-prone to error than

allocating data on the heap). In general, the variables are not intended to pass information between different types of typemaps. However, this can be done if you realize that local names have the argument number appended to them. For example, you could do this:

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

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

In this case, the `$argnum` variable is expanded into the argument number. Therefore, the code will reference the appropriate local such as `temp1` and `temp2`. It should be noted that there are plenty of opportunities to break the universe here and that accessing locals in this manner should probably be avoided. At the very least, you should make sure that the typemaps sharing information have exactly the same types and names.

10.15 C++ "this" pointer

All the rules discussed for Typemaps apply to C++ as well as C. However in addition C++ passes an extra parameter into every non-static class method -- the `this` pointer. Occasionally it can be useful to apply a typemap to this pointer (for example to check and make sure `this` is non-null before deferencing). Actually, C also has an the equivalent of the `this` pointer which is used when accessing variables in a C struct.

In order to customise the `this` pointer handling, target a variable named `self` in your typemaps. `self` is the name SWIG uses to refer to the extra parameter in wrapped functions.

For example, if wrapping for Java generation:

```
%typemap(check) SWIGTYPE *self %{
if (!$1) {
  SWIG_JavaThrowException(jenv, SWIG_JavaNullPointerException, "swigCPtr null");
  return $null;
}
%}
```

In the above case, the `$1` variable is expanded into the argument name that SWIG is using as the `this` pointer. SWIG will then insert the check code before the actual C++ class method is called, and will raise an exception rather than crash the Java virtual machine. The generated code will look something like:

```
  if (!arg1) {
    SWIG_JavaThrowException(jenv, SWIG_JavaNullPointerException,
      "invalid native object; delete() likely already called");
    return ;
  }
  (arg1)->wrappedFunction(...);
```

Note that if you have a parameter named `self` then it will also match the typemap. One work around is to create an interface file that wraps the method, but give the argument a name other than `self`.

10.16 Where to go for more information?

The best place to find out more information about writing typemaps is to look in the SWIG library. Most language modules define all of their default behavior using typemaps. These are found in files such as `python.swg`, `perl5.swg`, `tcl8.swg` and so forth. The `typemaps.i` file in the library also contains numerous examples. You should look at these files to get a feel for how to define typemaps of your own. Some of the language modules support additional typemaps and further information is available in the individual chapters for each target language. There you may also find more hands-on practical examples.