Multiple files and the SWIG library 4

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

The %include directive

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

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

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

The `%include` directive can process SWIG interface files, C header files, and C source files (provided they are sufficiently clean). When processing a C source file, SWIG will automatically declare all functions it finds as "extern". Thus, use of a header file may not be required in this case.

The %extern directive

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

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

// Now wrap a derived class
```

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

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

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

The %import directive

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

Including files on the command line

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

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

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

The SWIG library

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

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

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

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

Library example

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

```
%module example

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

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

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

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

Creating Library Files

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

tclsh.i

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

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

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

  /* Now initialize our functions */
```

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

  return TCL_OK;
}
%}
```

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

malloc.i

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

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

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

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

Placing the files in the library

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

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

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

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

Working with library files

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

Wrapping a library file

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

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

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

Checking out library files

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

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

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

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

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

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

```
%checkout myscript.pl
```

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

The world's fastest way to write a Makefile

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

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

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

Checking in library files

It is also possible to check files into the SWIG library. If you've made an interesting interface that

you would like to keep around, simply type :

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

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

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

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

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

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

Static initialization of multiple modules

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

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

... More declarations ...
```

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

More about the SWIG library

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