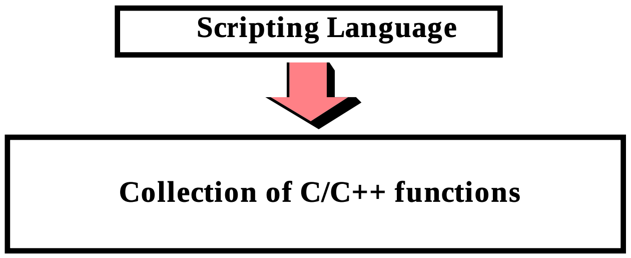
Scripting Languages 2

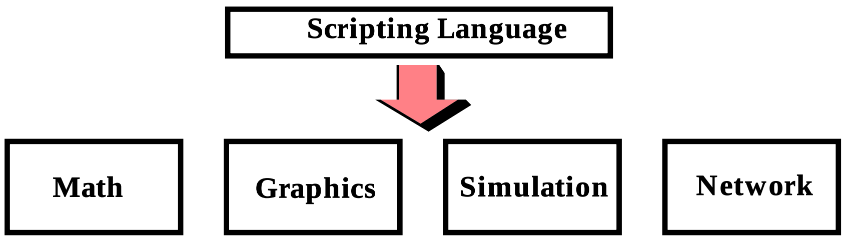
SWIG is all about using scripting languages with C/C++ to make flexible applications. This chapter provides a brief overview of several concepts and important aspects of this interface. Many of SWIG's potential users may not have considered using a scripting language before, so I hope that this chapter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

The two language view of the world

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



A real application might look more like this :



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

The two-language model of computing is extremely powerful because it exploits the strengths of each language. C/C++ can be used for maximal performance and complicated systems programming tasks. Scripting languages can be used for rapid prototyping, interactive debugging, scripting, and access to high-level data structures such as lists, arrays, and hash tables.

Will scripting languages make my C program inefficient?

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

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

or writing the same thing in Python :

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

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

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

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

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

How does a scripting language talk to C?

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

Wrapper functions

Suppose you have an ordinary C function like this :

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

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

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

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

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

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

```
int Wrap_Init(Tcl_Interp *interp) {
        Tcl_CreateCommand(interp, "fact", wrap_fact, (ClientData) NULL,
                                    (Tcl_CmdDeleteProc *) NULL);
        return TCL_OK;
}
```

When executed, Tcl will now have a new command called "`fact`" that you can use like any other Tcl command.

While the process of adding a new function to Tcl has been illustrated, the procedure is almost identical for Perl and Python. Both require special wrappers to be written and both need additional initialization code. Only the specific details are different.

Variable linking

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

```
double My_variable = 3.5;
```

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

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

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

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

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

Constants

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

Structures and classes

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

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

To provide access, the simplist approach is to transform a structure into a collection of accessor functions. For example :

```
struct Vector {
       Vector();
       ~Vector();
       double x,y,z;
};
```

can be transformed into the following set of functions :

```
Vector *new_Vector();
void delete_Vector(Vector *v);
double Vector_x_get(Vector *v);
double Vector_y_get(Vector *v);
double Vector_y_get(Vector *v);
void Vector_x_set(Vector *v, double x);
void Vector_y_set(Vector *v, double y);
void Vector_z_set(Vector *v, double z);
```

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

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

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

Shadow classes

As it turns out, it is possible to use the low-level accessor functions above to create something known as a "shadow" class. In a nutshell, a "shadow class" is a funny kind of object that gets created in a scripting language to access a C/C++ class (or struct) in a way that looks like the original structure (that is, it "shadows" the real C++ class). Of course, in reality, it's just a slick way of accessing objects that is more natural to most programmers. For example, if you have the following C definition :

```
class Vector {
public:
       Vector();
       ~Vector();
       double x,y,z;
};
```

A shadow classing mechanism would allow you to access the structure in a natural manner. For example, in Python, you might do the following,

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

while in Perl5, it might look like this :

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

and in Tcl :

```
Vector v
v configure -x 3 -y 4 -z 13
```

When shadow classes are used, two objects are at really work--one in the scripting language, and an underlying C/C++ object. Operations affect both objects equally and for all practical purposes, it appears as if you are simply manipulating a C/C++ object. However, the introduction of additional “objects” can also produce excessive overhead if working with huge numbers of objects in this manner. Despite this, shadow classes turn out to be extremely useful. The actual implementation is covered later.

Building scripting language extensions

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

Static linking

With static linking you rebuild the scripting language interpreter with extensions. The process usually involves compiling a short main program that adds your customized commands to the language and starts the interpreter. You then link your program with a library to produce a new executable. When using static linking, SWIG will provide a `main()` program for you so you usually just have to compile as follows (shown for Tcl) :

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

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

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

Shared libraries and dynamic loading

An alternative to static linking is to build a shared library. With this approach, you build a shared object file containing only the code related to your module (on Windows, this is the same as building a DLL). Unfortunately the process of building these modules varies on every single machine, but the procedure for a few common machines is show below :

```
# Build a shared library for Solaris
gcc -c example.c example_wrap.c -I/usr/local/include
ld -G example.o example_wrap.o -o example.so

# Build a shared library for Irix
gcc -c example.c example_wrap.c -I/usr/local/include
ld -shared example.o example_wrap.o -o example.so

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

To use your shared library, you simply use the corresponding command in the scripting language (load, import, use, etc...). This will import your module and allow you to start using it.

When working with C++ codes, the process of building shared libraries is more difficult--primarily due to the fact that C++ modules may need additional code in order to operate correctly. On most machines, you can build a shared C++ module by following the above procedures, but changing the link line to the following :

```
c++ -shared example.o example_wrap.o -o example.so
```

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

Linking with shared libraries

When building extensions as shared libraries, it is not uncommon for your extension to rely upon other shared libraries on your machine. In order for the extension to work, it needs to be able to find all of these libraries at run-time. Otherwise, you may get an error such as the following :

```
>>> import graph
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "/home/sci/data1/beazley/graph/graph.py", line 2, in ?
    import graphc
ImportError:  1101:/home/sci/data1/beazley/bin/python: rld: Fatal Error: cannot
successfully map soname 'libgraph.so' under any of the filenames /usr/lib/libgraph.so:/
lib/libgraph.so:/lib/cmplrs/cc/libgraph.so:/usr/lib/cmplrs/cc/libgraph.so:
>>>
```

What this error means is that the extension module created by SWIG depends upon a shared library called "`libgraph.so`" that the system was unable to locate. To fix this problem, there are a few approaches you can take.

- Set the UNIX environment variable `LD_LIBRARY_PATH` to the directory where shared libraries are located before running Python.
- Link your extension and explicitly tell the linker where the required libraries are located. Often times, this can be done with a special linker flag such as `-R`, `-rpath`, etc... This is not implemented in a standard manner so read the man pages for your linker to find out more about how to set the search path for shared libraries.
- Put shared libraries in the same directory as the executable. This technique is sometimes required for correct operation on non-Unix platforms.

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