The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

SWIG : An Easy to Use Tool For Integrating Scripting Languages with C and C++

David M. Beazley
University of Utah
Salt Lake City, Utah 84112

SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++

David M. Beazley
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
beazley@cs.utah.edu

Abstract

I present SWIG (Simplified Wrapper and Interface Generator), a program development tool that automatically generates the bindings between C/C++ code and common scripting languages including Tcl, Python, Perl and Guile. SWIG supports most C/C++ datatypes including pointers, structures, and classes. Unlike many other approaches, SWIG uses ANSI C/C++ declarations and requires the user to make virtually no modifications to the underlying C code. In addition, SWIG automatically produces documentation in HTML, LaTeX, or ASCII format. SWIG has been primarily designed for scientists, engineers, and application developers who would like to use scripting languages with their C/C++ programs without worrying about the underlying implementation details of each language or using a complicated software development tool. This paper concentrates on SWIG's use with Tcl/Tk.

1 Introduction

SWIG (Simplified Wrapper and Interface Generator) is a software development tool that I never intended to develop. At the time, I was trying to add a data analysis and visualization capability to a molecular dynamics (MD) code I had helped develop for massively parallel supercomputers at Los Alamos National Laboratory [Beazley, Lomdahl]. I wanted to provide a simple, yet flexible user interface that could be used to glue various code modules together and an extensible scripting language seemed like an ideal solution. Unfortunately there were constraints. First, I didn't want to hack up 4-years of code development trying to fit our MD code into yet another interface scheme (having done so several times already). Secondly, this code was routinely run on systems ranging from Connection Machines and Crays to workstations and I didn't want to depend on any one interface language—out of fear that it might not be supported on all of these platforms. Finally, the users were constantly adding new code and making modifications. I needed a flexible, yet easy to use system that did not get in the way of the physicists.

SWIG is my solution to this problem. Simply stated, SWIG automatically generates all of the code needed to bind C/C++ functions with scripting languages using only a simple input file containing C function and variable declarations. At first, I supported a scripting language I had developed specifically for use on massively parallel systems. Later I decided to rewrite SWIG in C++ and extend it to support Tcl, Python, Perl, Guile and other languages that interested me. I also added more data-types, support for pointers, C++ classes, documentation generation, and a few other features.

This paper provides a brief overview of SWIG with a particular emphasis on Tcl/Tk. However, the reader should remain aware that SWIG works equally well with Perl and other languages. It is not my intent to provide a tutorial or a user's guide, but rather to show how SWIG can be used to do interesting things such as adding Tcl/Tk interfaces to existing C applications, quickly debugging and prototyping C code, and building interface-language-independent C applications.

2 Tcl and Wrapper Functions

In order to add a new C or C++ function to Tcl, it is necessary to write a special "wrapper" function that parses the function arguments presented as ASCII strings by the Tcl interpreter into a representation that can be used to call the C function. For example, if you wanted to add the factorial function to Tcl, a wrapper function 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;
}
```

In addition to writing the wrapper function, a user will also need to write code to add this function to the Tcl interpreter. In the case of Tcl 7.5, this could be done by writing an initialization function to be called when the extension is loaded dynamically. While writing a wrapper function usually is not too difficult, the process quickly becomes tedious and error prone as the number of functions increases. Therefore, automated approaches for producing wrapper functions are appealing–especially when working with a large number of C functions or with C++ (in which case the wrapper code tends to get more complicated).

3 Prior Work

The idea of automatically generating wrapper code is certainly not new. Some efforts such as Itcl++, Object Tcl, or the XS language included with Perl5, provide a mechanism for generating wrapper code, but require the user to provide detailed specifications, type conversion rules, or use a specialized syntax [Heidrich, Wetherall, Perl5]. Large packages such as the Visualization Toolkit (vtk) may use their own C/C++ translators, but these almost always tend to be somewhat special purpose (in fact, SWIG started out in this manner) [vtk]. If supporting multiple languages is the ultimate goal, a programmer might consider a package such as ILU [Janssen]. Unfortunately, this requires the user to provide specifications in IDL–a process which is unappealing to many users. SWIG is not necessarily intended to compete with these approaches, but rather is designed to be a no-nonsense tool that scientists and engineers can use to easily add Tcl and other scripting languages to their own applications. SWIG is also very different than Embedded Tk (ET) which also aims to simplify code development [ET]. Unlike ET, SWIG is designed to integrate C functions into Tcl/Tk as opposed to integrating Tcl/Tk into C programs.

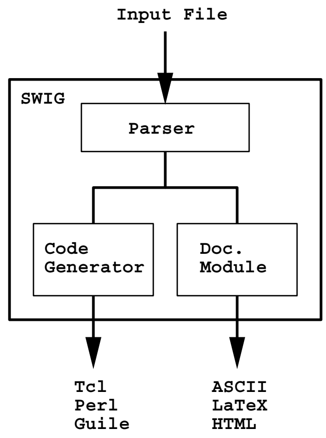

Figure 1: SWIG organization.

4 A Quick Tour of SWIG

4.1 Organization

Figure 1 shows the structure of SWIG. At the core is a YACC parser for reading input files along with some utility functions. To generate code, the parser calls about a dozen functions from a generic language class to do things like write a wrapper function, link a variable, wrap a C++ member function, etc... Each target language is implemented as a C++ class containing the functions that emit the resulting C code. If an "empty" language definition is given to SWIG, it will produce no output. Thus, each language class can be implemented in almost any manner. The documentation system is implemented in a similar manner and can currently produce ASCII, LaTeX, or HTML output. As output, SWIG produces a C file that should be compiled and linked with the rest of the code and a documentation file that can be used for later reference.

4.2 Interface Files

As input, SWIG takes a single input file referred to as an "interface file." This file contains a few SWIG specific directives, but otherwise contains ANSI C function and variable declarations. Unlike the approach in [Heidrich], no type conversion rules are needed and all declarations are made using familiar ANSI C/C++ prototypes. The following code shows an interface file for wrapping a few C file I/O and memory management functions.

```
/* File : file.i */
%module fileio
%{
#include <stdio.h>
%}

FILE *fopen(char *filename, char *type);
int  fclose(FILE *stream);
typedef unsigned int size_t
size_t  fread(void *ptr, size_t size,
               size_t nobj, FILE *stream);
size_t  fwrite(void *ptr, size_t size,
                size_t nobj,FILE *stream);
void *malloc(size_t nbytes);
void  free(void *);
```

The `%module` directive sets the name of the initialization function. This is optional, but is recommended if building a Tcl 7.5 module. Everything inside the `%{`, `%}` block is copied directly into the output, allowing the inclusion of header files and additional C code. Afterwards, C/C++ function and variable declarations are listed in any order. Building a new Tcl module is usually as easy as the following :

```
unix > swig -tcl file.i
unix > gcc file_wrap.c -I/usr/local/include
unix > ld -shared file_wrap.o -o Fileio.so
```

4.3 A Tcl Example

Newly added functions work like ordinary Tcl procedures. For example, the following Tcl script copies a file using the binary file I/O functions added in the previous example :

```
proc filecopy {name1 name2} {
    set buffer [malloc 8192];
    set f1 [fopen $name1 r];
    set f2 [fopen $name2 w];
    set nbytes [fread $buffer 1 8192 $f1];
    while {$nbytes > 0} {
        fwrite $buffer 1 $nbytes $f2;
        set nbytes [fread $buffer 1 8192 $f1];
    }
    fclose $f1;
    fclose $f2;
    free $buffer
}
```

4.4 Datatypes and Pointers

SWIG supports the basic datatypes of `int`, `short`, `long`, `float`, `double`, `char`, and `void` as well as signed and unsigned integers. SWIG also allows derived types such as pointers, structures, and classes, but these are all encoded as pointers. If an unknown type is encountered, SWIG assumes that it is a complex datatype that has been defined earlier. No attempt is made to figure out what data that datatype actually contains or how it should be used. Of course, this this is only possible since SWIG's mapping of complex types into pointers allows them to be handled in a uniform manner. As a result, SWIG does not normally need any sort of type-mapping, but `typedef` can be used to map any of the built-in datatypes to new types if desired.

SWIG encodes pointers as hexadecimal strings with type-information. This type information is used to provide a run-time type checking mechanism. Thus, a typical SWIG pointer looks something like the following :

```
_1008e614_Vector_p
```

If this pointer is passed into a function requiring some other kind of pointer, SWIG will generate a Tcl error and return an error message. The NULL pointer is represented by the string "NULL". The SWIG run-time type checker is saavy to typedefs and the relationship between base classes and derived classes in C++. Thus if a user specifies

```
typedef double Real;
```

the type checker knows that `Real *` and `double *` are equivalent (more on C++ in a minute). From the point of view of other Tcl extensions, SWIG pointers should be seen as special "handles" except that they happen to contain the pointer value and its type.

To some, this approach may seem horribly restrictive (or error prone), but keep in mind that SWIG was primarily designed to work with existing C applications. Since most C programs pass complex datatypes around by reference this technique works remarkably well in practice. Run time type-checking also eliminates most common crashes by catching stupid mistakes such as using a wrong variable name or forgetting the "$" character in a Tcl script. While it is still possible to crash Tcl by forging a SWIG pointer value (or making a call to buggy C code), it is worth emphasizing that existing Tcl extensions may also crash if given an invalid handle.

4.5 Global Variables and Constants

SWIG can install global C variables and constants using Tcl's variable linkage mechanism. Variables may also be declared as "read only" within the Tcl interpreter. The following example shows how variables and constants can be added to Tcl :

```
// SWIG file with variables and constants
%{
%}

// Some global variables
extern  int My_variable;
extern  char *default_path;
extern  double My_double;

// Some constants
#define  PI     3.14159265359
#define  PI_4   PI/4.0
enum colors {red,blue,green};
const int SIZEOF_VECTOR = sizeof(Vector);

// A read only variable
%readonly
extern int Status;
%readwrite
```

4.6 C++ Support

The SWIG parser can handle simple C++ class definitions and supports public inheritance, virtual functions, static functions, constructors and destructors. Currently, C++ translation is performed by politely tranforming C++ code into C code and generating wrappers for the C functions. For example, consider the following SWIG interface file containing a C++ class definition:

```
%module tree
%{
#include "tree.h"
%}

class Tree {
public:
   Tree();
  ~Tree();
   void  insert(char *item);
   int   search(char *item);
   int   remove(char *item);
static void print(Tree *t);
};
```

When translated, the class will be access used the following set of functions (created automatically by SWIG).

```
Tree *new_Tree();
void delete_Tree(Tree *this);
void Tree_insert(Tree *this, char *item);
int  Tree_search(Tree *this, char *item);
int  Tree_remove(Tree *this, char *item);
void Tree_print(Tree *t);
```

All C++ functions wrapped by SWIG explicitly require the `this` pointer as shown. This approach has the advantage of working for all of the target languages. It also makes it easier to pass objects between other C++ functions since every C++ object is simply represented as a SWIG pointer. SWIG does not support function overloading, but overloaded functions can be resolved by renaming them with the SWIG `%name` directive as follows:

```
class List {
public:
              List();
%name(ListMax) List(int maxsize);
...
}
```

The approach used by SWIG is quite different than that used in systems such as Object Tcl or vtk [vtk, Wetherall]. As a result, users of those systems may find it to be confusing. However, It is important to note that the modular design of SWIG allows the user to completely redefine the output behavior of the system. Thus, while the current C++ implementation is quite different than other systems supporting C++, it would be entirely possible write a new SWIG module that wrapped C++ classes into a representation similar to that used by Object Tcl (in fact, in might even be possible to use SWIG to produce the input files used for Object Tcl).

4.7 Multiple Files and Code Reuse

An essential feature of SWIG is its support for multiple files and modules. A SWIG interface file may include another interface file using the "`%include`" directive. Thus, an interface for a large system might be broken up into a collection of smaller modules as shown

```
%module package
%{
#include "package.h"
%}

%include geometry.i
%include memory.i
%include network.i
%include graphics.i
%include physics.i

%include wish.i
```

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.