SWIG for developers in a hurry

[Arpan Sen](arpansen@gmail.com) (arpansen@gmail.com)
Lead Engineer
electronic design automation industry

Skill Level: Intermediate

Date: 23 Apr 2012

SWIG is a nifty open source tool that lets you integrate `C/C++` code with just about any mainstream scripting language. Among other things, it exposes the code base to a wider audience, improves testability, and lets a portion of your Ruby code base run off high-performance `C/C++` modules.

SWIG installation

This article used SWIG version 2.0.4 (see Resources for a link to the download site). To build and install SWIG, you follow the typical open source installation process, entering the following commands at a command prompt:

```
tar xvzf swig-2.0.4.tar.gz
./configure –prefix=/your/swig/install/path
make
make install
```

Note that the path given to the prefix must be an absolute path.

`C` and `C++` are widely recognized (and rightly so) as being the platform of choice for creating high-performance code. A common request to developers is for them to expose the `C/C++` code from a scripting language interface, and this is where the Simplified Wrapper and Interface Generator (SWIG) comes in. SWIG lets you expose your `C/C++` code to a wide range of scripting languages, including Ruby, Perl, Tcl, and Python. This article uses Ruby as the scripting interface of choice for exposing `C/C++` functionality. To follow along with this article, you should have an intermediate knowledge of both `C/C++` and Ruby.

SWIG is a great tool in several scenarios, including:

- Providing a scripting interface to `C/C++` code, making it easier for users
- Adding extensions to your Ruby code or replacing existing modules with high-performance alternatives
- Providing the ability to do unit and integration testing of your code using a scripting environment

- Developing a graphical user interface in, say, TK and integrating it with the `C/C++` back end

In addition, SWIG is a lot easier to debug than firing up the GNU Debugger every time.

Ruby environment variables

SWIG generates wrapper `C/C++` code that requires ruby.h to compile correctly. Check your Ruby installation for ruby.h: It's a recommended practice to have the environment variable RUBY_INCLUDE point to the folder that contains ruby.h and RUBY_LIB point to the path that contains the Ruby library.

Hello World with SWIG

As input, SWIG expects a file containing ANSI `C/C++` declarations and SWIG directives. I call this input file the *SWIG interface file.* It's important to remember that SWIG needs only as much information as necessary to generate wrapper code. The interface file typically has an **.i* or **.swg* extension. Here's the first extension file, test.i:

```
%module test
%constant char* Text = "Hello World with SWIG"
```

Run this code with SWIG:

```
swig –ruby test.i
```

The command line in the second snippet generates a file called *test_wrap.c* in the current folder. Now, you need to create a shared library out of this `c` file. Here's the command line:

```
bash$ gcc –fPIC –c test_wrap.c –I$RUBY_INCLUDE
bash$ gcc –shared test_wrap.o –o test_wrap.so –lruby –L$RUBY_LIB
```

That's it. You're all set, so just fire up the interactive Ruby shell (IRB), and enter `require 'test_wrap'` to check out the Ruby `Test` module and its contents. Here's the Ruby side of the extension:

```
irb(main):001:0> require 'test_wrap'
=> true
irb(main):002:0> Test.constants
=> ["Text"]
irb(main):003:0> Test:: Text
=> "Hello World with SWIG"
```

SWIG can be used to generate a variety of language extensions, just run `swig –help` to check out all the available options. For Ruby, you enter `swig –ruby <interface file>`; for Perl, you use `swig –perl <interface file>`.

You can also use SWIG to generate `C++` code: You just use `–c++` in the command line. In the previous example, running `swig –c++ –ruby test.i` generates a file called *test_wrap.cxx* in the current folder.

The SWIG basics

SWIG interface file syntax is a superset of `C`. Indeed, SWIG processes its input files through a custom `C` preprocessor. In addition, SWIG operations in an interface file are controlled through special directives (`%module`, `%constant`, and so on) that are preceded by the percent sign (`%`). SWIG interfaces also let you define blocks that begin with `%{` and end with `%}`. Everything between `%{` and `%}` is copied verbatim to the generated wrapper file.

More about module names

It's possible to define a deeply nested module like `rubytest::test34::example` by specifying the same as `%module "rubytest::test34::example`. Yet another option is to have `%module example` in the interface code and prefix it with `rubytest::test34` from the command line as follows:

```
bash$ swig –c++ –ruby –prefix "rubytest::test34" test.i
```

SWIG interface files must begin with a `%module` declaration—for example, `%module` *`module-name`*, where *module-name* is the name of the target language extension module. If the target language is Ruby, this is akin to creating a Ruby module. It is possible to override the module name by providing the command-line option `–module` *`module-name-modified`*: In this case, the target language module name is (you guessed it) *module-name-modified.* Now, let's move on to constants.

The module initialization function

SWIG has a `%init` special directive that's used to define module initialization functionality. The code defined inside the `%{ … %}` block that follows `%init` is what gets called when the module loads. Here's the code:

```
%module test
%constant char* Text = "Hello World with SWIG"
%init %{
printf("Initialization etc. gets done here\n");
%}
```

Now, restart IRB. Here's what you get once the module is loaded:

```
irb(main):001:0> require 'test'
Initialization etc. gets done here
=> true
```

SWIG constants

`C/C++` constants can be defined in multiple ways in an interface file. To verify whether the same constants have been exposed to the Ruby module, just type `<module-`

`name>.constants` at the IRB prompt when you have the shared library loaded. You can define constants in any of the following ways:

- Using `#define` in an interface file
- Using `enum`s
- Using the `%constant` directive

Note that Ruby constants must begin with an uppercase letter. So, if your interface file has a declaration such as `#define pi 3.1415`, SWIG automatically corrects it to `#define Pi 3.1415` and generates a warning message in the process:

```
bash$ swig –c++ –ruby test.i
test.i(3) : Warning 801: Wrong constant name (corrected to 'Pi')
```

The following example contains a lot of constants. Run it as `swig –ruby test.i`:

```
%module test
#define S_Hello "Hello World"
%constant double PI = 3.1415
enum days {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

Listing 1 shows the output from SWIG.

Listing 1. Exposing C enums to Ruby: What went wrong?

```
test_wrap.c: In function `Init_test':
test_wrap.c:2147: error: `Sunday' undeclared (first use in this function)
test_wrap.c:2147: error: (Each undeclared identifier is reported only once
test_wrap.c:2147: error: for each function it appears in.)
test_wrap.c:2148: error: `Monday' undeclared (first use in this function)
test_wrap.c:2149: error: `Tuesday' undeclared (first use in this function)
test_wrap.c:2150: error: `Wednesday' undeclared (first use in this function)
test_wrap.c:2151: error: `Thursday' undeclared (first use in this function)
test_wrap.c:2152: error: `Friday' undeclared (first use in this function)
test_wrap.c:2153: error: `Saturday' undeclared (first use in this function)
```

Oops: What happened? If you open test_wrap.c (Listing 2), you can see the problem.

Listing 2. Understanding SWIG-generated code for enums

```
  rb_define_const(mTest, "Sunday", SWIG_From_int((int)(Sunday)));
  rb_define_const(mTest, "Monday", SWIG_From_int((int)(Monday)));
  rb_define_const(mTest, "Tuesday", SWIG_From_int((int)(Tuesday)));
  rb_define_const(mTest, "Wednesday", SWIG_From_int((int)(Wednesday)));
  rb_define_const(mTest, "Thursday", SWIG_From_int((int)(Thursday)));
  rb_define_const(mTest, "Friday", SWIG_From_int((int)(Friday)));
  rb_define_const(mTest, "Saturday", SWIG_From_int((int)(Saturday)));
```

SWIG is creating Ruby constants out of Sunday, Monday, and the rest, except that the original `enum` declaration for `day` is missing in the generated file. The simplest way to solve this problem is to put the `enum` code inside the `%{ … %}` block so that the generated file knows about the enumeration constants, as Listing 3 shows.

Listing 3. Exposing C enums to Ruby the right way

```
%module test
#define S_Hello "Hello World"
%constant double PI = 3.1415
enum days {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
%{
enum days {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
%}
```

Note that the `enum` declaration alone will not make the enumeration constants available to the scripting environment: You need both the `C` code inside `%{ … %}` and the `enum` declaration in the interface file.

Introducing the %inline special directive

Listing 3 is ugly— code duplication for `enum` is uncalled for. To remove the duplication, you need to use the `%inline` SWIG directive. The `%inline` directive inserts all of the code in the `%{ … %}` block that follows it verbatim into the interface file so that both the SWIG preprocessor and the `C` compiler are satisfied. Listing 4 shows the revamped code, with the `enum` now being used with `%inline`.

Listing 4. Using %inline directives to reduce code duplication

```
%module test
#define S_Hello "Hello World"
%constant double PI = 3.1415
%inline %{
enum days {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
%}
```

%include is an even cleaner approach

In a complex enterprise environment, there would likely be `C/C++` headers defining global variables and constants that you want to expose to the scripting framework. Using `%include <header.h>` and `%{ #include <header.h> %}` in the interface file solves the problem of repeat declarations of all elements in the header. Listing 5 shows the code.

Listing 5. Using the %include directive

```
%module test
%include "header.h"

%{
#include "header.h"
%}
```

The `%include` directive also works on `C/C++` source files. When used with source files, SWIG automatically declares all functions as `extern`.

Enough of constants: Let's expose some functions

The simplest way to begin learning SWIG is to declare some `C` function in the interface file, have it defined in some source file, and link the corresponding object file while creating the shared library. This first example shows a function that computes the factorial of a number:

```
%module test
unsigned long factorial(unsigned long);
```

Here's the `C` code that I compiled into factorial.o and linked while creating test.so:

```
unsigned long factorial(unsigned long n) {
   return n == 1 ? 1 : n * factorial(n - 1);
}
```

Listing 6 shows the Ruby interface.

Listing 6. Testing the code from Ruby

```
irb(main):001:0> require 'test'
=> true
irb(main):002:0> Test.factorial(11)
=> 39916800
irb(main):003:0> Test.factorial(34)
=> 0
```

Factorial 34 failed, because unsigned `long` was not of sufficient width to capture the result.

Ruby-to-C/C++ mapping for variables

Let's begin with simple global variables. Note that `C/C++` global variables are not really global to Ruby: You can only access them as module attributes. Add the following global variables in a `C` file, and have the sources linked much the same way as you did for functions. SWIG automatically generates the setter and getter methods of these variables for you. Here's the `C` code:

```
int global_int1;
long global_long1;
float global_float1;
double global_double1;
```

Listing 7 shows the interface for the same.

Listing 7. Exposing the C interface to Ruby

```
%module test
%inline %{
extern int global_int1;
extern long global_long1;
extern float global_float1;
extern double global_double1;
%}
```

Now, load the corresponding Ruby module to verify the addition of the setter and getter methods:

```
irb(main):003:0> Test.methods
[…"global_float1", "global_float1=", "global_int1", "global_int1=", "global_long1",
   "global_long1=", "global_double1", "global_double1=", …]
```

Accessing the variables is now simple:

```
irb(main):004:0> Test.global_long1 = 4327911
=> 4327911
irb(main):005:0> puts Test.global_long1
=> 4327911
```

Of particular interest is what Ruby converted `int`, `long`, `float`, and `double` to. See Listing 8.

Listing 8. Type mapping between Ruby and C/C++

```
irb(main):009:0> Test::global_long1.class
=> Fixnum
irb(main):010:0> Test::global_int1.class
=> Fixnum
irb(main):011:0> Test::global_double1.class
=> Float
irb(main):012:0> Test::global_float1.class
=> Float
```

Mapping structures and classes from C++ to Ruby

Exposing structures and classes to Ruby follows on the same line as `C/C++` plain old data types. You simply declare the structure and related methods in the interface file. Listing 9 declares a simple `Point` structure and a function to compute the distance between them. On the Ruby side, you create a new `Point` as `Test::Point.new` and invoke the compute distance as `Test.distance_between`. The `distance_between` function is defined in a separate `C++` source file that's linked with the module shared library. Here's the SWIG interface code:

Listing 9. Exposing structures and the related interface to Ruby

```
%module test

%inline %{
typedef struct Point {
  int x;
  int y;
};
extern float distance_between(Point& p1, Point& p2);
%}
```

Listing 10 shows the Ruby usage.

Listing 10. Verifying the C/C++ functionality from Ruby

```
irb(main):002:0> a = Test::Point.new
=> #<Test::Point:0x2d04260>
irb(main):003:0> a.x = 10
=> 10
irb(main):004:0> a.y = 20
=> 20
irb(main):005:0> b = Test::Point.new
=> #<Test::Point:0x2cce668>
irb(main):006:0> b.x = 20
=> 20
irb(main):007:0> b.y = 10
=> 10
irb(main):008:0> Test.distance_between(a, b)
=> 14.1421356201172
```

This use model should make it amply clear why SWIG is an excellent tool to have handy while setting up a unit or integration test framework for your code base.

%defaultctor and other attributes

If you check out the default values of the *x* and *y* coordinates of a point, you see that they come out as 0. This is no coincidence. SWIG is generating default constructors for your structure. You could turn this behavior off by specifying `%nodefaultctor Point;` in the interface file. Listing 11 shows how.

Listing 11. No default constructor for C++ structures

```
%module test
%nodefaultctor Point;
%inline %{
typedef struct Point {
  int x;
  int y;
};
%}
```

You now need to provide an explicit constructor for the `Point` structure, as well. Otherwise, you would see the following code:

```
irb(main):005:0> a = Test::Point.new
TypeError: allocator undefined for Test::Point
        from (irb):5:in `new'
        from (irb):5
```

It's possible to make every structure define its constructor explicitly by specifying `%nodefaultctor;` in the interface file. SWIG also defines the `%nodefaultdtor` directive for similar functionality in destructors.

C++ inheritance and the Ruby interface

For simplicity's sake, say you have two `C++` classes—`Base` and `Derived`—in the interface file. SWIG is fully aware that `Derived` is derived from `Base`. From a Ruby perspective, you can simply use `Derived.new` and safely expect that the created object knows that it's derived from `Base`. Listing 12 shows the Ruby test code; nothing specific need be done on the `C++` or SWIG interface side.

Listing 12. The SWIG interface handles C++ inheritance

```
irb(main):003:0> a = Test::Derived.new
=> #<Test::Derived:0x2d06270>
irb(main):004:0> a.instance_of? Test::Derived
=> true
irb(main):005:0> a.instance_of? Test::Base
=> false
irb(main):006:0> Test::Derived < Test::Base
=> true
irb(main):007:0> Test::Derived > Test::Base
=> false
irb(main):008:0> a.is_a? Test::Derived
=> true
irb(main):009:0> a.is_a? Test::Base
=> true
```

The handling is not that smooth with `C++` multiple inheritance. If `Derived` were inherited from `Base1` and `Base2`, then the default SWIG behavior is to simply ignore `Base2`. Here's the message you would get from SWIG:

```
Warning 802: Warning for Derived d: base Base2 ignored.
   Multiple inheritance is not supported in Ruby.
```

Frankly, SWIG cannot be faulted, because Ruby does not support multiple inheritance. For SWIG to work, you need to pass the `–minherit` option in the command line:

```
bash$ swig -ruby -minherit -c++ test.i
```

It's important to understand how SWIG handles multiple inheritance. The derived class in `C++` corresponds to a class in Ruby that is derived from neither `Base1` nor `Base2`. Instead, `Base1` and `Base2` code is re-factored into modules and included in `Derived`. This is what in Ruby terminology is called *mixin.* Listing 13 shows the pseudocode of what's happening.

Listing 13. Simulating multiple inheritance with Ruby

```
class Base1
  module Impl
  # Define Base1 methods here
  end
  include Impl
end

class Base2
  module Impl
  # Define Base2 methods here
  end
  include Impl
end

class Derived
  module Impl
  include Base1::Impl
  include Base2::Impl
  # Define Derived methods here
  end
  include Impl
end
```

Let's verify the claim from the Ruby interface. The `included_modules` method does the trick for you, as Listing 14 shows.

Listing 14. Multiple modules as part of the Ruby class

```
irb> Test::Derived.included_modules
=> [Test::Derived::Impl, Test::Base::Impl, Test::Base2::Impl, Kernel]

irb> Test::Derived < Test::Base
=> nil

irb> Test::Derived < Test::Base2
=> nil
```

Note that the class hierarchy test fails (as it should), but to the application developer, the functionality of `Base` and `Base2` is still available via the `Derived` class.

Pointers and the Ruby interface

Ruby does not have an equivalent for pointers, so what happens to `C/C++` methods that accept or return pointers? This brings us to one of the most important challenges for a system like SWIG, whose primary job is to convert (or *marshal,* as they say) data types between the source and destination languages. Consider the following `C` function:

```
void addition(const int* n1, const int* n2, int* result) {
  *result = *n1 + *n2;
}
```

To solve this problem, SWIG introduces the notion of *type maps.* You have the flexibility of mapping which Ruby type you want to map to for `int*`, `float*`, and the rest. Thankfully though, SWIG has already done most of the boilerplate work for you. So here's the simplest interface you could possibly have for addition:

```
%module Test
%include typemaps.i
void addition (int* INPUT, int* INPUT, int* OUTPUT);

%{ extern void addition(int*, int*, int*); %}
```

Now, try the code from Ruby as `Test::addition(1, 2)`. You should be able to see the result. To understand more about what's happening here, look in the lib/ruby folder. SWIG is using the `int* INPUT` syntax to convert the underlying pointer into an object. The SWIG syntax for mapping a type from Ruby to `C/C++` is:

```
%typemap(in) int* {
  … type conversion code from Ruby to C/C++
}
```

Likewise, the type conversion code from `C/C++` to Ruby is:

```
%typemap(out) int* {
  … type convesion code from C/C++ to Ruby
}
```

Type maps don't just come handy for pointers: You can use them for just about any data type conversion between Ruby and `C/C++`.

Conclusion

In this article, you learned how to expose `C/C++` constants; variables, including structures and classes; functions; and enumerations to the Ruby interface. Along the way, you picked up SWIG directives like `%module`, `%init`, `%constant`, `%inline`,

`%include`, and `%nodefaultctor`. SWIG offers a lot more; be sure to check out the excellent PDF document that comes with the SWIG documentation for more details.

Resources

Learn

- Check out the SWIG documentation.
- AIX and UNIX developerWorks zone: The AIX and UNIX zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX? Visit the New to AIX and UNIX page to learn more.
- Technology bookstore: Browse the technology bookstore for books on this and other technical topics.

Get products and technologies

- Download the latest version of SWIG.
- Try out IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.

Discuss

- Follow developerWorks on Twitter.
- Participate in developerWorks blogs and get involved in the developerWorks community.
- Get involved in the My developerWorks community.
- Participate in the AIX and UNIX® forums:
  - AIX Forum
  - AIX Forum for developers
  - Cluster Systems Management
  - Performance Tools Forum
  - Virtualization Forum
  - More AIX and UNIX Forums

About the author

Arpan Sen

Arpan Sen is a lead engineer working on the development of software in the electronic design automation industry. He has worked on several flavors of UNIX, including Solaris, SunOS, HP-UX, and IRIX as well as Linux and Microsoft Windows for several years. He takes a keen interest in software performance-optimization techniques, graph theory, and parallel computing. Arpan holds a post-graduate degree in software systems.