FREE eBook

LEARNING swig

#swig

Table of Contents

About

Chapter 1: Getting started with swig

Remarks

SWIG (Simplified Wrapper and Interface Generator) is a tool for wrapping C and C++ code in a variety of target languages, allowing C/C++ APIs to be used in other languages.

SWIG parses header files and generates code in a manner dependent on the target language. The code generation can be controlled by the developer in the *SWIG interface file* as well as through command line options.

In the interface file, the developer tells SWIG what to wrap and how. SWIG has its own preprocessor system and many special directives to control how data, classes and functions are wrapped in the target language. Some of these directives are general and others are specific to the target language.

Central to how SWIG functions is the *typemap*. Typemaps are rules that specify how types are marshaled between the C code and the target language. Typemaps can be applied globally to everything in the interface file or locally on a case by case basis. They can also be customized if necessary.

Once SWIG is run on the interface file, it produces a C or C++ file which is the wrapper. This file should be compiled and linked with the C/C++ program or static library the wrapper is meant to interface with to produce a shared library. That library in turn is used by the target language.

RTFM

It cannot be emphasized enough that SWIG already comes with an excellent documentation manual. This is very detailed on the one hand, covers installation, and features many concrete examples in the form of code snippets, including a complete "hello world" SWIG example.

But most importantly, it also explains 1.7 How to avoid reading the manual:

> If you hate reading manuals, glance at the "Introduction" which contains a few simple examples. These examples contain about 95% of everything you need to know to use SWIG. After that, simply use the language-specific chapters as a reference. The SWIG distribution also comes with a large directory of examples that illustrate different topics.

Examples

Installation or Setup

Detailed instructions on getting swig set up or installed.

Hello World

A minimal example of using SWIG.

HelloWorld.i, the SWIG interface file

```
%module helloworld    //the name of the module SWIG will create
%{                    //code inside %{...%} gets inserted into the wrapper file
#include "myheader.h" //helloworld_wrap.cxx includes this header
%}

%include "myheader.h"   //include the header for SWIG to parse
```

Then, in the command line

```
swig -c++ -java HelloWorld.i
```

which means we are wrapping C++ (as opposed to C) with Java as the target language as specified by HelloWorld.i. This will produce a C++ file, helloworld_wrap.cxx, which has the wrapper code. This file should be compiled and linked against whatever code the wrapper is supposed to interface with (e.g., a static library) to produce a shared library. With some languages, as with Java in our example, additional code will be generated - in our case, there will be at least one Java class file.

Read Getting started with swig online: https://riptutorial.com/swig/topic/7608/getting-started-with-swig

Chapter 2: Introduction to Typemaps

Introduction

Typemaps are the very heart of what SWIG does. When you want to pass data between languages the behaviours for doing so depend upon the type that SWIG sees. The power of typemaps is that the chunks of code are applied many times.

SWIG itself includes many useful typemaps in the core library it is supplied with, e.g. for primitive types, C++ standard library containers, boost etc. so often you won't even need to write any typemaps to expose your code, however that list is by no means complete.

Syntax

- %typemap(NAME) TYPENAME %{ CODE %}
- %typemap(NAME,OPTION=VALUE) TYPENAME %{ CODE %}
- %typemap(NAME) TYPENAME VARIABLENAME %{ CODE %}
- %typemap(NAME) TYPENAME (LOCALVARTYPE LOCALVARNAME) %{ CODE %}

Parameters

Parameter	Details
NAME	The name of the typemap defines its role in generating a module. `in` and `out` are common and used for input (to C++ or C function calls from Python/Java etc.) and output (i.e. return values from C or C++ to Pyton/Java)
TYPENAME	Each typemap gets applied to one or more matching types. These need to be listed here. Type qualifiers (e.g. const) matters.
CODE	Every typemap needs to convert between the C or C++ type and the corresponding type in the wrapped language. You will need to write code to do that in your custom typemaps, typically making use of special variables that get substituted in. E.g. `$input` inside an `in` typemap represents the value from Python/Java, `$result` in an `out` typemap is the thing being returned to Python/Java. `$1` in both in/out typemaps represents the C or C++ variable of the type used to match the typemap. So for `in` typemaps you would assign to `$1` and for `out` you would read from it. (See multi-argument typemaps for more details on why it is a number)
VARIABLENAME	Typemaps are matched from most specific to least specific in general. You can define a typemap that will only match function arguments with specific names by using this optional form. (See typemap

Parameter	Details
	matching for more details)
(LOCALVARTYPE LOCALVARNAME)	Sometimes, particulary for `in` typemaps it's useful to be able to declare extra local variables to hold objects around a call. This optional syntax allows us do do that. By using this syntax instead of writing it in `CODE` the scope is altered, but more importantly the variable can be automatically renamed by SWIG to avoid clashes if a function has two arguments using the same typemap.
OPTION=VALUE	The behaviour of some typemaps can be influenced by setting extra options using this syntax. For example a `in` typemap can be made to take no input from the target language by setting `numinputs=0`, in which case the typemap is expected to fill the input implicitly. (A common case for this might be to set something to `NULL`, or fill it from a global value)

Examples

Basic typemap - Python

Given the following custom Boolean type we want to wrap:

```
typedef char MYBOOL;
#define TRUE 1
#define FALSE 0
```

A simple approach might be to write the following typemaps in our SWIG interface:

```
%typemap(in) MYBOOL %{
  // $input is what we got passed from Python for this function argument
  $1 = PyObject_IsTrue($input);
  // $1 is what will be used for the C or C++ call and we are responsible for setting it
%}

%typemap(out) MYBOOL %{
  // $1 is what we got from our C or C++ call
  $result = PyBool_FromLong($1);
  // $result is what gets given back to Python and we are responsible for setting it
%}
```

With these typemaps, SWIG will insert our code into the generated wrapper every time it sees a MYBOOL passed into or out of a function call.

Read Introduction to Typemaps online: https://riptutorial.com/swig/topic/9289/introduction-to-typemaps

Credits

S. No	Chapters	Contributors
1	Getting started with swig	BenK, Community, m7thon
2	Introduction to Typemaps	Flexo