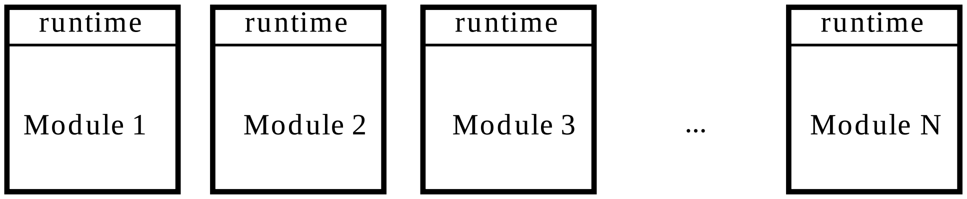
Advanced Topics 11

Creating multi-module packages

SWIG can be used to create packages consisting of many different modules. However, there are some technical aspects of doing this and techniques for managing the problem.

Runtime support (and potential problems)

All SWIG generated modules rely upon a small collection of functions that are used during runtime. These functions are primarily used for pointer type-checking, exception handling, and so on. When you run SWIG, these functions are included in the wrapper file (and declared as static). If you create a system consisting of many modules, each one will have an identical copy of these runtime libraries :



SWIG used with multiple modules (default behavior)

This duplication of runtime libraries is usually harmless since there are no namespace conflicts and memory overhead is minimal. However, there is serious problem related to the fact that modules do not share type-information. This is particularly a problem when working with C++ (as described next).

Why doesn’t C++ inheritance work between modules?

Consider for a moment the following two interface files :

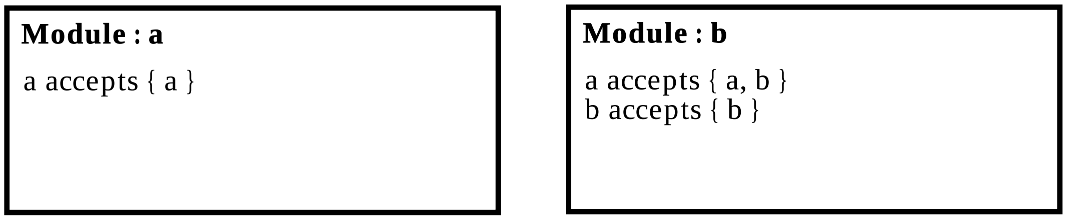
```
// File : a.i
%module a

// Here is a base class
class a {
public:
       a();
       ~a();
       void foo(double);
};
```

```
// File : b.i
%module b

// Here is a derived class
%extern a.i                 // Gets definition of base class

class b : public a {
public:
       bar();
};
```

When compiled into two separate modules, the code does not work properly. In fact, you get a type error such as the following :

```
[beazley@guinness shadow]$ python
Python 1.4 (Jan 16 1997)  [GCC 2.7.2]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from a import *
>>> from b import *
>>> # Create a new "b"
>>> b = new_b()
>>> # Call a function in the base class
...
>>> a_foo(b,3)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: Type error in argument 1 of a_foo. Expected _a_p.
>>>
```

However, from our class definitions we know that “b” is an “a” by inheritance and there should be no type-error. This problem is directly due to the lack of type-sharing between modules. If we look closely at the module modules created here, they look like this :



Two SWIG modules with type information

The type information listed shows the acceptable values for various C datatypes. In the “a” module, we see that “a” can only accept instances of itself. In the “b” module, we see that “a” can accept both “a” and “b” instances--which is correct given that a “b” is an “a” by inheritance.

Unfortunately, this problem is inherent in the method by which SWIG makes modules. When we made the “a” module, we had no idea what derived classes might be used at a later time. However, it’s impossible to produce the proper type information until after we know all of the derived classes. A nice problem to be sure, but one that can be fixed by making all modules share a single copy of the SWIG run-time library.

The SWIG runtime library

To reduce overhead and to fix type-handling problems, it is possible to share the SWIG run-time functions between multiple modules. This requires the use of the SWIG runtime library which is optionally built during SWIG installation. To use the runtime libraries, follow these steps :

1. Build the SWIG run-time libraries. The `SWIG1.1/Runtime` directory contains a makefile for doing this. If successfully built, you will end up with 6 files that are usually installed in `/usr/local/lib`.

```
libswigtcl.a                # Tcl library (static)
libswigtcl.so               # Tcl library (shared)
libswigpl.a                 # Perl library (static)
libswigpl.so                # Perl library (shared)
libswigpy.a                 # Python library (static)
libswigpy.so                # Python library (shared)
```

Note that certain libraries may be missing due to missing packages or unsupported features (like dynamic loading) on your machine.

2. Compile all SWIG modules using the `-c` option. For example :

```
% swig -c -python a.i
% swig -c -python b.i
```

The `-c` option tells SWIG to omit runtime support. It's now up to you to provide it separately--which we will do using our libraries.

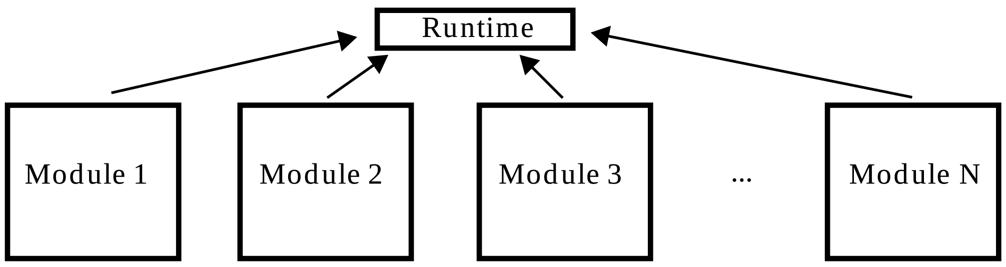
3. Build SWIG modules by linking against the appropriate runtime libraries.

```
% swig -c -python a.i
% swig -c -python b.i
% gcc -c a_wrap.c b_wrap.c -I/usr/local/include
% ld -shared a_wrap.o b_wrap.o -lswigpy  -o a.so
```

or if building a new executable (static linking)

```
% swig -c -tcl -ltclsh.i a.i
% gcc a_wrap.c -I/usr/local/include -L/usr/local/lib -ltcl -lswigtcl -lm -o mytclsh
```

When completed you should now end up with a collection of modules like this :



In this configuration, the runtime library manages all datatypes and other information between modules. This management process is dynamic in nature--when new modules are loaded, they contribute information to the run-time system. In the C++ world, one could incrementally load classes as needed. As this process occurs, type information is updated and base-classes learn about derived classes as needed.

A few dynamic loading gotchas

When working with dynamic loading, it is critical to check that only one copy of the run-time library is being loaded into the system. When working with `.a` library files, problems can sometimes occur so there are a few approaches to the problem.

1. Rebuild the scripting language executable with the SWIG runtime library attached to it. This is actually, fairly easy to do using SWIG. For example :

```
%module mytclsh
%{

static void *__embedfunc(void *a) { return a};
%}

void *__embedfunc(void *);
%include tclsh.i
```

Now, run SWIG and compile as follows :

```
% swig -c -tcl mytclsh.i
% gcc mytclsh_wrap.c -I/usr/local/include -L/usr/local/lib -ltcl -lswigtcl -ldl -lm \
      -o tclsh
```

This produces a new executable "`tclsh`" that contains a copy of the SWIG runtime library. The weird `__embedfunc()` function is needed to force the functions in the runtime library to be included in the final executable.

To make new dynamically loadable SWIG modules, simply compile as follows :

```
% swig -c -tcl example.i
% gcc -c example_wrap.c -I/usr/local/include
% ld -shared example_wrap.o -o example.so
```

Linking against the `swigtcl` library is no longer necessary as all of the functions are now included in the `tclsh` executable and will be resolved when your module is loaded.

2. Using shared library versions of the runtime library

If supported on your machine, the runtime libraries will be built as shared libraries (indicated by a `.so`, `.sl`, or `.dll` suffix). To compile using the runtime libraries, you link process should look something like this :

```
% ld -shared swigtcl_wrap.o -o libswigtcl.so             # Irix
% gcc -shared swigtcl_wrap.o -o libswigtcl.so            # Linux
% ld -G swigtcl_wrap.o -o libswigtcl.so                  # Solaris
```

In order for the libswigtcl.so library to work, it needs to be placed in a location where the dynamic loader can find it. Typically this is a system library directory (ie. `/usr/local/lib` or `/usr/lib`).

When running with the shared libary version, you may get error messages such as the following

```
Unable to locate libswigtcl.so
```

This indicates that the loader was unable to find the shared libary at run-time. To find shared libaries, the loader looks through a collection of predetermined paths. If the `libswigtcl.so` file is not in any of these directories, it results in an error. On most machines, you can change the loader search path by changing the Unix environment variable `LD_LIBRARY_PATH`. For example :

```
% setenv LD_LIBRARY_PATH .:/home/beazley/packages/lib
```

A somewhat better approach is to link your module with the proper path encoded. This is typically done using the '-rpath' or '-R' option to your linker (see the man page). For example :

```
% ld -shared example_wrap.o example.o -rpath /home/beazley/packages/lib \
      -L/home/beazley/packages/lib -lswigtcl.so -o example.so
```

The `-rpath` option encodes the location of shared libraries into your modules and gets around having to set the `LD_LIBRARY_PATH` variable.

If all else fails, pull up the man pages for your linker and start playing around.

Dynamic Loading of C++ modules

Dynamic loading of C++ modules presents a special problem for many systems. This is because C++ modules often need additional supporting code for proper initialization and operation. Static constructors are also a bit of a problem.

While the process of building C++ modules is, by no means, and exact science, here are a few rules of thumb to follow :

- Don't use static constructors if at all possible (not always avoidable).
- Try linking your module with the C++ compiler using a command like 'c++ -shared'. This often solves alot of problems.
- Sometimes it is necessary to link against special libraries. For example, modules compiled with g++ often need to be linked against the `libgcc.a`, `libg++.a`, and `libstdc++.a` libraries.
- Read the compiler and linker man pages over and over until you have them memorized (this may not help in some cases however).
- Search articles on Usenet, particularly in `comp.lang.tcl`, `comp.lang.perl`, and `comp.lang.python`. Building C++ modules is a common problem.

The SWIG distribution contains some additional documentation about C++ modules in the Doc directory as well.

Inside the SWIG type-checker

The SWIG runtime type-checker plays a critical role in the correct operation of SWIG modules. It not only checks the validity of pointer types, but also manages C++ inheritance, and performs proper type-casting of pointers when necessary. This section provides some insight into what it does, how it works, and why it is the way it is.

Type equivalence

SWIG uses a name-based approach to managing pointer datatypes. For example, if you are using a pointer like "`double *`", the type-checker will look for a particular string representation of that datatype such as "`_double_p`". If no match is found, a type-error is reported.

However, the matching process is complicated by the fact that datatypes may use a variety of different names. For example, the following declarations

```
typedef double   Real;
typedef Real *   RealPtr;
typedef double   Float;
```

define two sets of equivalent types :

```
{double, Real, Float}
{RealPtr, Real *}
```

All of the types in each set are freely interchangable and the type-checker knows about the relationships by managing a table of equivalences such as the following :

```
double    => { Real, Float }
Real      => { double, Float }
Float     => { double, Real }
RealPtr   => { Real * }
Real *    => { RealPtr }
```

When you declare a function such as the following :

```
void foo(Real *a);
```

SWIG first checks to see if the argument passed is a "`Real *`". If not, it checks to see if it is any of the other equivalent types (`double *`, `RealPtr`, `Float *`). If so, the value is accepted and no error occurs.

Derived versions of the various datatypes are also legal. For example, if you had a function like this,

```
void bar(Float ***a);
```

The type-checker will accept pointers of type `double ***` and `Real ***.` However, the type-checker does not always capture the full-range of possibilities. For example, a datatype of '`RealPtr **`' is equivalent to a '`Float ***`' but would be flagged as a type error. If you encounter this kind of problem, you can manually force SWIG to make an equivalence as follows:

```
// Tell the type checker that ‘Float_ppp’ and ‘RealPtr_pp’ are equivalent.
%init %{
       SWIG_RegisterMapping(“Float_ppp”,”RealPtr_pp”,0);
%}
```

Doing this should hardly ever be necessary (I have never encountered a case where this was necessary), but if all else fails, you can force the run-time type checker into doing what you want.

Type-equivalence of C++ classes is handled in a similar manner, but is encoded in a manner to support inheritance. For example, consider the following classes hierarchy :

```
class A { };
class B : public A { };
class C : public B { };
class D {};
class E : public C, public D {};
```

The type-checker encodes this into the following sets :

```
A => { B, C, E }                   ”B isa A, C isa A, E isa A”
B => { C, E }                      “C isa B, E isa B”
C => { E }                         “E isa C”
D => { E }                         “E isa D”
E => { }
```

The encoding reflects the class hierarchy. For example, any object of type “A” will also accept objects of type B,C, and E because these are all derived from A. However, it is not legal to go the other way. For example, a function operating on a object from class E will not accept an object from class A.

Type casting

When working with C++ classes, SWIG needs to perform proper typecasting between derived and base classes. This is particularly important when working with multiple inheritance. To do this, conversion functions are created such as the following :

```
void *EtoA(void *ptr) {
       E *in = (E *) ptr;
       A *out = (A *) in;          // Cast using C++
       return (void *) out;
}
```

All pointers are internally represented as void *, but conversion functions are always invoked when pointer values are converted between base and derived classes in a C++ class hierarchy.

Why a name based approach?

SWIG uses a name-based approach to type-checking for a number of reasons :

- One of SWIG’s main uses is code development and debugging. In this environment, the type name of an object turns out to be a useful piece of information in tracking down problems.
- In languages like Perl, the name of a datatype is used to determine things like packages

and classes. By using datatype names we get a natural mapping between C and Perl.
- I believe using the original names of datatypes is more intuitive than munging them into something completely different.

An alternative to a name based scheme would be to generate type-signatures based on the structure of a datatype. Such a scheme would result in perfect type-checking, but I think it would also result in a very confusing scripting language module. For this reason, I see SWIG sticking with the name-based approach--at least for the foreseeable future.

Performance of the type-checker

The type-checker performs the following steps when matching a datatype :

1. Check a pointer against the type supplied in the original C declaration. If there is a perfect match, we're done.
2. Check the supplied pointer against a cache of recently used datatypes.
3. Search for a match against the full list of equivalent datatypes.
4. If not found, report an error.

Most well-structured C codes will find an exact match on the first attempt, providing the best possible performance. For C++ codes, it is quite common to be passing various objects of a common base-class around between functions. When base-class functions are invoked, it almost always results in a miscompare (because the type-checker is looking for the base-type). In this case, we drop down to a small cache of recently used datatypes. If we've used a pointer of the same type recently, it will be in the cache and we can match against it. For tight loops, this results in about 10-15% overhead over finding a match on the first try. Finally, as a last resort, we need to search the internal pointer tables for a match. This involves a combination of hash table lookup and linear search. If a match is found, it is placed into the cache and the result returned. If not, we finally report a type-mismatch.

As a rule of thumb, C++ programs require somewhat more processing than C programs, but this seems to be avoidable. Also, keep in mind that performance penalties in the type-checker don't necessarily translate into big penalties in the overall application. Performance is most greatly affected by the efficiency of the target scripting language and the types of operations your C code is performing.