SWIG and Fortran

Seth R Johnson

Contents

Overview

This chapter describes how to create interfaces to C and C++ data and functions in the target language of Fortran, a long-lived scientific programming language. The original motivation for adding the Fortran language to SWIG was to provide an automated means of adapting massively parallel scientific codes to modern solvers and GPU-accelerated kernels in the Trilinos numerical library package (ForTrilinos). But adding Fortran as a SWIG target language has the potential to simplify and accelerate numerous existing Fortran codes that do not require advanced numerical solvers: it is now tremendously simple to generate Fortran library modules from existing C and C++ libraries.

SWIG differs from other attempts to couple C/C++ and Fortran in that it is designed to provide *C and C++* functionality to *Fortran*, and not to generically make these two languages (or others like Python) interoperable. SWIG only parses C and C++ code; it does not parse Fortran code or generate C/C++ interfaces to Fortran libraries. SWIG assumes that you, the library developer, have an existing, working C/C++ interface that you wish to *adapt* to the Fortran target language. This adaptation may include tweaks for ease of use or familiarity for Fortran users, but it does not require that your library be developed around a central interface language. This is in contrast to other existing cross-language interoperability tools such as Babel.

Fundamental concepts

The purpose of running SWIG with the `-fortran` language option is to generate a Fortran module that can be used by other Fortran code. This module contains automatically generated code that provides a Fortran interface to existing C or C++ interfaces. SWIG generates a `.f90` Fortran module file, and a separate `_wrap.c` or `_wrap.cxx` file of implementation code that the module must link against.

The C/C++ `_wrap` file contains simple, flat, C-linkage interface functions that provide access to arbitrarily complicated C/C++ data and functions. The conversion may be as simple as casting one integer type to another, or as complicated as allocating a piece of memory and calling a function to encode a complex object. These interface functions, which are namespaced with a `swigc_` prefix, translate the C/C++ data (classes, enumerations) into simple ANSI C types (integers, structs).

The C function signature of those interfaces is translated to private `interface` declarations in the Fortran module to `bind(C)` functions. These interfaces use only data types compatible with Fortran 2003's `ISO_C_BINDING` features. Those bound interface functions are called by SWIG-generated Fortran wrapper code that converts C-compatible data types to native Fortran data types.

These two layers of translation allow complex C++ datatypes to be translated to complex Fortran data. For example, `std::string` objects can be automatically converted to Fortran `character(len=:), allocatable` variables; and `const std::vector<double>&` references can be accepted from `real(c_double), dimension(:), target` Fortran data.

Besides translating simple data types, SWIG can generate "proxy classes" in Fortran from C++ classes. These thin Fortran 2003 "derived type" definitions bind a C++ class instance to a Fortran-friendly object equivalent.

Nomenclature

The terminology in C/C++ and Fortran is different enough to be potentially confusing to a user not intimately familiar with both languages. The author is more familiar with C++ than Fortran but has endeavored to use the correct Fortran terms when describing the Fortran implementation. The following table presents some equivalent concepts and names in the two languages:

C/C++	Fortran
struct/class	derived type
function	procedure
member function	type-bound procedure
function that returns void	subroutine
function that returns non-void	function
overloaded function	generic interface
floating point number	real
fundamental type	intrinsic type
derived type	extended type
function parameters	dummy arguments

Identifiers

C and C++ have different rules for identifiers (i.e. variable names, function names, class names) than Fortran. The following restrictions apply to Fortran that do not apply to C and C++:

- Names are **case insensitive**
- Names may not begin with an underscore
- Names may be no longer than 63 characters

SWIG automatically renames identifiers that start with a leading underscore. It keeps a symbol table of publicly accessible Fortran identifiers (as their lower-cased, renamed versions) and warns about and ignores duplicate names.

There is also no "namespace" concept in Fortran 2003 aside from defining procedures and types in separate modules. (Fortran 2008 supports submodules, but these are more akin to private namespaces inside a translation unit in C++.) Keep in mind that the flexible `%rename` directive can be used to adjust the symbolic names created in SWIG.

Running SWIG

Suppose that we have a SWIG interface file `example.i` with the following contents:

```
/* File: example.i */
%module forexample

%{
```

```
/* include header */
#include "cexample.h"
%}

%include "cexample.h"
```

where `cexample.h` contains the simple function declaration:

```
int fact(int n);
```

To generate SWIG Fortran wrappers for this file, run

```
$ swig -fortran example.i
```

and SWIG will create two files: a C interface file containing something like

```
/* SNIP */
/* include header */
#include "cexample.h"
/* SNIP */
SWIGEXPORT int swigc_fact(int const *farg1) {
  int fresult;
  int arg1;
  int result;

  arg1 = *farg1;
  result = (int)fact(arg1);
  fresult = result;
  return fresult;
}
/* SNIP */
```

and a Fortran interface file with something like:

```
module forexample
 use, intrinsic :: ISO_C_BINDING
 implicit none
 public :: fact
private
interface
 function swigc_fact(farg1) &
   bind(C, name="swigc_fact") &
   result(fresult)
  use, intrinsic :: ISO_C_BINDING
  integer(C_INT) :: fresult
  integer(C_INT), intent(in) :: farg1
 end function
end interface
contains
 function fact(n) &
```

```
   result(swig_result)
  use, intrinsic :: ISO_C_BINDING
  integer(C_INT) :: swig_result
  integer(C_INT), intent(in) :: n
  integer(C_INT) :: fresult
  integer(C_INT) :: farg1
  farg1 = n
  fresult = swigc_fact(farg1)
  swig_result = fresult
 end function
end module forexample
```

The above contrived example uses different names for the `%module` declaration, the interface `.i` file, and the wrapped C header `.h` file to illustrate how these inputs affect the output file names and properties:

- The `%module forexample` declaration in the SWIG interface file resulted in the file names `forexample.f90` and the name in `module forexample`.
- The file name `example.i` resulted in the C wrapper file by default being named `example_wrap.c`.
- The `#include` command was explicitly inserted into the C wrapper file `example_wrap.c`.
- The `%include` command in the `.i` file directed SWIG to parse the header file `cexample.h` and generate an interface for the function declaration that it discovered. The typical convention is to keep these names consistent: almost without exception, the module name `%module forexample` should be reflected in the file name as `forexample.i`.

In the generated C wrapper code above, `int swigc_fact(int const *farg1)` is the wrapper code generated by SWIG to provide a Fortran-compatible interface with the C function `fact`; the `function swigc_fact(farg1)` interface in Fortran is the exact equivalent of that C function; and the "public" function `fact` in the `contains` section of the Fortran module is the Fortran proxy function generated by SWIG.

Note that since this function takes and returns simple data types, the C and Fortran wrapper functions have some code that could be easily simplified by hand. (A compiler with optimization enabled automatically does this, in fact.) However, for more complex data types, SWIG shows its power by generating complex expressions that seamlessly translate between C and Fortran data types without requiring user intervention.

Compiling a simple Fortran program

Several examples are provided in the SWIG source code under `Examples/fortran/`. In the `barefunctions` example, the Fortran `main` program can be compiled using the following sequence of commands:

```
swig -fortran -c++ bare.i
$CXX -c bare_wrap.cxx
$CXX -c bare.cxx -o barecxx.o
$FC -c bare.f90
$FC runme.f90 bare.o bare_wrap.o barecxx.o -lstdc++ -o run.exe
```

Note that since this was a C++ program, the `-c++` option must be passed to SWIG and `-lstdc++` must be passed to the final link command. Also note that the three middle commands, which create object files, can be executed in any order. Because the `swig` command generates both `bare_wrap.cxx` and `bare.f90`, it must be first. The final executable command, which links against all three generated object files, must be last.

Compiling more complex Fortran/C/C++ programs

The figure below shows how C++, SWIG, and Fortran code can be integrated into libraries and linked to form executables. The file icons are user-written files; circles are executables; flat cylinders are generated on-disk files; and the final box is the executable. The arrow signifies "generates" or "is used by."

We provide CMake modules and commands to simplify this process; again, see the example directories for usage instructions.

Basic Fortran/C data type interoperability

The Fortran SWIG module relies on Fortran 2003's C interoperability features, both the `ISO_C_BINDING` intrinsic module and the specifications of the standard. Every effort has been made to conform to the standard in the translation layer between C++ and Fortran and to eliminate potential pitfalls of interoperability.

We anticipate that future extensions of Fortran/C interoperability will increase the capability of the SWIG wrapper interface. (For example, the Fortran ISO technical specification TS29113 will greatly extend the types of arrays and pointers that can be passed between C and Fortran.)

However, many features of C and C++ are outside the scope of Fortran's interoperability features. Even some features that *are* interoperable,such as enumerations and structs, have capabilities that do not map between the two languages. With this SWIG module we attempt to extend the Fortran/C++ mapping as much as possible, keeping in mind that Fortran and C are inherently different languages.

Fundamental types

SWIG maps ISO C types to Fortran types using the `ISO_C_BINDING` intrinsic module. The fundamental types fully supported by C, Fortran, and SWIG are:

C type	Fortran type
`signed char`	`integer(C_SIGNED_CHAR)`
`short`	`integer(C_SHORT)`
`int`	`integer(C_INT)`
`long`	`integer(C_LONG)`
`long long`	`integer(C_LONG_LONG)`
`size_t`	`integer(C_SIZE_T)`
`float`	`real(C_FLOAT)`
`double`	`real(C_DOUBLE)`
`char`	`character(C_CHAR)`

Pointers and references to the fundamental types are returned as scalar Fortran pointers.

Note that because the C return value does not contain any information about the shape of the data being pointed to, it is not possible to directly construct an array from a pointed-to value. However, advanced typemaps can be constructed (and indeed are provided with the SWIG Fortran standard library) that *can* return that information or extend the Fortran interface to obtain the additional information needed to return an array pointer.

Other integer types

Fortran has no intrinsic unsigned datatypes, so the specification says to treat `unsigned` datatypes as their `signed` counterparts. Note that this means `unsigned char` will be wrapped as a Fortran integer by default.

No checking for negativity or boundedness is done when converting the datatypes. In part this is because intentionally out-of-range values (e.g., `static_cast<size_t>(-1)`) are often used as sentinels.

A more complete set of typemaps for the full set of integer types available in `<stdint.h>` can be used by `%include <cstdint>`.

Boolean/logical values

The astute reader may notice the omission of `C_BOOL` from the above table. Because of the different treatment of booleans in C and Fortran, guaranteeing the sizes of the `bool` are equivalent in the two languages does *not* guarantee the equivalence of their values. See this discussion topic for details of the subtle compatibility, but in brief, Fortran's `.true.` is defined by having the least significant bit set to `1`, whereas C defines it as any nonzero value. So the value `2` would be `true` in C but `false` in Fortran. A special typemap inserts wrapper code to explicitly convert booleans between the two languages.

Characters

Since `char*`, `const char[]`, etc. typically signify character strings in C and C++, the default behavior of these is to convert to native Fortran strings (see the Strings section). To restore the "fundamental" behavior of a character type – i.e., you want to make a `char *` return result map to `character(C_CHAR), pointer` – you can call an internal macro and apply it to the particular function or argument you need:

```
typedef char NativeChar;
FORT_FUND_TYPEMAP(NativeChar, "character(C_CHAR)")
%apply NativeChar* { char * get_my_char_ptr };

char* get_my_char_ptr();
```

Pointers and references

C pointers and mutable references are treated as Fortran pointers. Suppose a C function that returns a pointer to an array element at a given index:

```
double* get_array_element(int x);
```

This generates the following Fortran interface:

```
function get_array_element(x) &
      result(swig_result)
  use, intrinsic :: ISO_C_BINDING
  real(C_DOUBLE), pointer :: swig_result
  integer(C_INT), intent(in) :: x
  type(C_PTR) :: fresult
  integer(C_INT) :: farg1

  farg1 = x
  fresult = swigc_get_array_element(farg1)
  call c_f_pointer(fresult, swig_result)
end function
```

To set the element at array index 2 to the value 512,

```
real(C_DOUBLE), pointer :: rptr
rptr => get_array_element(2)
rptr = 512.0d0
```

Note, and this is **very important**, that a function returning a pointer must not be *assigned*; the *pointer assignment* operator `=>` must be used.

Mutable references are treated identically. However, *const* references to fundamental types are treated as values:

```
const double& get_const_array_element(int x);
```

will generate

```
function get_const_array_element(x) &
    result(swig_result)
  use, intrinsic :: ISO_C_BINDING
  real(C_DOUBLE) :: swig_result
  integer(C_INT), intent(in) :: x
  real(C_DOUBLE) :: fresult
  integer(C_INT) :: farg1

  farg1 = x
  fresult = swigc_get_const_array_element(farg1)
  swig_result = fresult
end function
```

which must be called like

```
real(C_DOUBLE) :: rval
rval = get_const_array_element(2)
```

Combining the two examples, you could copy the value of element 3 to element 2 with the following code block:

```
real(C_DOUBLE), pointer :: rptr
rptr => get_array_element(2)
rptr = get_const_array_element(3)
```

First the pointer is assigned, then the pointed-to data is assigned.

Strings

A long-standing difficulty with Fortran/C interaction has been the two languages' representation of character strings. The size of a C string is determined by counting the number of characters until a null terminator `\0` is encountered. Shortening a string requires simply placing the null terminator earlier in the storage space. In contrast, the historical Fortran string is a character array sized at compile time: representing a smaller string at run time is done by filling the storage with trailing blanks. The Fortran intrinsic `LEN_TRIM` returns the length of a string without trailing blanks, and the `TRIM` function is used if necessary to return a string with those trailing blanks removed. Of course, this definition of a string means `'foo'` and `'foo '` are equivalent.

Starting with Fortran 90, strings with an unambiguous size can be dynamically allocated:

```
character(kind=C_CHAR, len=:), allocatable :: mystring
allocate(character(kind=C_CHAR, len=123) :: mystring)
```

and the length is given by `LEN(mystring)`.

SWIG injects small helper functions that convert between strings and arrays of characters, which are then passed through the interface layer to C. Because the actual Fortran string length is passed to C during this process, character arrays with the null character can be converted to byte objects without unexpected string truncation.

The default `char*` typemaps assume that both the input and output are standard null-terminated C strings on the C++ side, and a variable-length string on the Fortran side (i.e. any trailing blanks are intentional). Note that by using null-terminated strings, if a Fortran string has null characters embedded in it, the string will be truncated when read by C. Thus the function as written is *not* suitable for passing binary data between C and Fortran. (See byte strings for how to do this.)

If a function `char* to_string(float f);` emits a `malloc`'d string value, and the output is to be wrapped by SWIG, use the `%newobject` feature to avoid memory leaks:

```
%apply const char* NATIVE { char* to_string };
%newobject to_string;
char* to_string(float f);
```

The Fortran-to-C string translation performs the following steps:

1. Allocates a character array of `len(string) + 1`
2. Copies the string contents into that array and sets the final character to `C_NULL_CHAR`
3. Saves the C pointer to the character array using `C_LOC` and the size to a small `SwigArrayWrapper` struct
4. Passes this struct to the C wrapper code, which uses the data pointer.

The C-to-Fortran string translation is similar:

1. Use `strlen` to save the string length to `SwigArrayWrapper.size`, and save the pointer to the data; return this struct to Fortran
2. Call `C_F_POINTER` to reinterpret the opaque C pointer as a character array
3. Allocate a new string with a length determined by the `size` member
4. Copy the character array to the new string
5. If the `%newobject` feature applies, call the C-bound `free` function.

The intermediate step of allocating and copying an array is required not only to add a null terminator but also because the Fortran 2003's interoperability specifications prohibit using `C_LOC` on variables with length type parameters. Thus the standard `character(len=*)` type cannot be natively passed to C.

Improved support for the various character typemaps and representations (as in the standard SWIG `<cstring.i>` which provides `%cstring_bounded_output`) could be implemented in a later version of SWIG.

Arrays

SWIG supports a subset of direct Fortran array translation. If a single-dimensional array size is explicitly specified in a C function's signature, the corresponding argument will be an explicit-shape Fortran array.

One note of caution is that occasionally arrays will be defined using nontrivial C expressions rather than explicit integers. Even though these can be evaluated by C at compile time, the unevaluated expression cannot be propagated into the Fortran wrapper code. SWIG checks whether the expression is a combination of base-10 numbers and the simple arithmetic expressions `+-*/`; if so, it is allowable. Otherwise, a warning is emitted and the array is ignored.

```
int global_data1[8]; /* OK */
int global_data2[];  /* OK */
int global_data3[sizeof(int)];  /* WARN AND IGNORE */
```

Byte strings

SWIG provides a two-argument typemap for converting fixed-length byte sequences, useful for passing buffers of binary data. This typemap searches for two consecutive function arguments called `(char *STRING, size_t LENGTH)`; but like any other SWIG typemap it can be applied to other argument names as well:

```
%apply (char *STRING, size_t LENGTH) { (const char *buf, size_t len) }
void send_bytes(int dst, const char *buf, size_t len);
```

can be used in Fortran as:

```
call send_bytes(123, "these are" // c_null_char // " some bytes")
```

The function will be passed the actual length of the byte string (9 + 1 + 10) in addition to the raw data, including the embedded null character. Compare this to

```
void send_bytes(int dst, const char *buf);
```

which would treat buf as a C string, use strlen to find its length, and truncate it at the first null character (for a length of 9).

Classes and structs

Each wrapped C++ class or struct produces a corresponding *derived type* in the wrapper code that holds a type(C_PTR) that points to an existing C++ class instance. SWIG seamlessly translates these derived type instances to and from their C++ equivalent.

SWIG wraps classes and structs identically. After all, in C++, the only difference between a struct and a class is the default *access specifier*: public for struct and private for class. As with the rest of SWIG, only public methods and data are wrapped.

Unlike many other SWIG target languages, the Fortran-wrapped classes are *strongly typed*: the compiler enforces type checking between data types and function arguments.

Ignored or unimplemented forward-declared classes

Some functions may include references or pointers to classes that are not wrapped by Fortran proxy functions. In these cases, an opaque derived type called SwigUnknownClass will be generated and used as a placeholder for the argument or return value. These could theoretically be passed between wrapped SWIG functions, although no type checking will be performed to ensure that the unknown classes are the correct types.

Enumerations

Fortran 2003 implements C enumerations using the ENUM, BIND(C) statement. These enumerators are simply a set of loosely grouped compile-time integer constants that are guaranteed to be compatible with C enumerators. Unlike C++, all enumerators in Fortran are anonymous.

To associate a C enumeration name with the Fortran generated wrappers, SWIG generates an additional enumeration with the C class name and a dummy value of -1. The enumeration generated from the C code

```
enum MyEnum {
    RED = 0,
    GREEN,
    BLUE,
    BLACK = -1
};
```

looks like:

```
enum, bind(c)
 enumerator :: MyEnum = -1
 enumerator :: RED = 0
 enumerator :: GREEN = RED + 1
 enumerator :: BLUE = GREEN + 1
 enumerator :: BLACK = -1
end enum
```

These enumerators are treated as standard C integers in the C wrapper code code. In the Fortran wrapper code, procedures that use the enumeration use the type `integer(kind(MyEnum))` to clearly indicate what enum type is required.

Some C++ enumeration definitions cannot be natively interpreted by a Fortran compiler (e.g. `FOO = 0x12,` or `BAR = sizeof(int),`), so these are defined in the C++ wrapper code and *bound* in the Fortran wrapper code:

```
integer(C_INT), protected, public, &
   bind(C, name="swigc_FOO") :: FOO
```

The `%enumerator` and `%noenumerator` directives can be used to explicitly enable and disable treatment of a C++ `enum` as a Fortran enumerator. Disabling the enumerator feature causes the value to be wrapped as externally-bound C integers.

Function pointers

It is possible to pass function pointers both from C to Fortran and from Fortran to C using SWIG. Currently, function pointer variables simply generate opaque `type(C_FUNPTR)` objects, and it is up to the user to convert to a Fortran procedure pointer using c_f_procpointer:

```
subroutine CallIt(cp) bind(c)
  use, intrinsic :: iso_c_binding
  type(c_funptr), intent(in) :: cp
  abstract interface
    subroutine Add_Int(i) bind(c)
      import
      integer(c_int), intent(inout) :: i
    end subroutine Add_Int
  end interface
  procedure(Add_Int), pointer :: fp
  integer(c_int) :: j

  call c_f_procpointer(cp, fp)
  j = 1
  call fp(j)
end subroutine CallIt
```

See the `funcptr` example in SWIG for an example of the callback functionality in practice.

Currently function pointers only work with user-created C-linkage functions as described below, but we plan to extend function callbacks so that data can be translated through wrapper functions.

Another planned extension for function pointers is to automatically generate the necessary *abstract interface* code required by Fortran to interpret the function pointer.

Handles and other oddities

Most combinations of pointers and references (such as `int**`, `int* const*`, `int*[3]`, `int*&`) are treated as opaque C pointers. They can be passed through the Fortran/C interface but currently have no special meaning or operations in generated Fortran code.

```
double** get_handle();
```

becomes

```
function get_handle() &
    result(swig_result)
  use, intrinsic :: ISO_C_BINDING
  type(C_PTR) :: swig_result
end function
```

Similarly, member function pointers (bound to a member function of a particular class instance) are supported as opaque Fortran objects.

Basic C/C++ features

This section describes the wrapper and proxy code generated by C and C++ language features.

Functions

Functions in C/C++ are *procedures* in Fortran. Their arguments correspond directly between the two languages: one argument in the C code requires one argument in the Fortran proxy. (Two exceptions are that C arguments can be ignored by swig using the `%typemap(in, numinputs=0)` directive in SWIG, and that SWIG supports multiple-argument typemaps.) A function in C/C++ with a `void` return value will translate to a `subroutine` in Fortran, and a function returning anything else will yield a Fortran `function`.

Each function in SWIG has a unique "symbolic name" or *symname* bound to it. The *symname* must be compatible with C linkage, and thus namespaces, templates, and overloads are incorporated into the symname, but a symname is often just the same as the bare function name.

SWIG will generate a wrapper function in the C++ file named `swigc_$symname`, where `$symname` is replaced with the symname. A corresponding private `BIND(C)` interface statement will be generated in the Fortran interface module. This wrapper function is responsible for converting the function's arguments and return value to and from Fortran-compatible datatypes and calling the C++ function. It also implements other optional features such as exception handling.

In the Fortran module, SWIG generates a public procedure `$symname` that translates native Fortran data types to and from the C interface datatypes. This interface, and not the `swigc_$symname` bound function, is the one used by Fortran application codes.

Function overloading

There is an important exception to the naming scheme described above: function overloading, when two or more free functions share a name but have different arguments. For each overloaded function signature, SWIG generates a private procedure with a unique symname. These procedures are then combined under a *separate module procedure* that is given a public interface with the original symbolic name. For example, an overloaded free function `myfunc` in C++ will generate two private procedures and add an interface to the module specification:

```
public :: myfunc
interface myfunc
 module procedure myfunc__SWIG_0, myfunc__SWIG_1
end interface
```

It should be noted that a function that returns `void` cannot be overloaded with a function that returns anything else: generic interfaces must be either all subroutines or all functions. Use SWIG's `%ignore` statement to hide one or the other:

```
void cannot_overload(int x);
int  cannot_overload(int x, int y);
%ignore cannot_overload(int x);
```

Global variables

Global variables in SWIG are wrapped with "getter" and "setter" functions. In the case of a global C++ variable

```
namespace foo {
extern int global_counter;
}
```

SWIG will generate functions with interfaces

```
subroutine set_global_counter(value0)
  use, intrinsic :: ISO_C_BINDING
  integer(C_INT), intent(in) :: value0
end subroutine
```

and

```
function get_global_counter() &
  result(swigf_result)
  use, intrinsic :: ISO_C_BINDING
  integer(C_INT) :: swigf_result
end function
```

Although no type conversion is needed for simple integers, other global data types would require special wrapper code in these functions.

Currently, global C/Fortran-compatible variables are treated the same as C++ data, but in the future we plan to expand the `%bindc` feature to directly wrap

```
extern "C" {
extern int global_counter_c;
}
```

as a C-bound common block variable bound

```
integer(C_INT), bind(C, name="global_counter_c") :: global_counter_c
```

Global constants

Global constant variables (whether declared in C++ headers with `const` or in a SWIG wrapper with `%constant`) of native types can be wrapped as Fortran parameters:

```
%parameter approx_pi;
const double approx_pi = 3.1416;
```

will be translated to

```
 real(C_DOUBLE), parameter, public :: approx_pi = 3.1416_C_DOUBLE
```

If the variable is defined in the header file and is a simple integer, this feature will be enabled by default. It can be explicitly enabled or disabled using the `%parameter` and `%noparameter` directives.

Global constants that have the feature disabled will be wrapped as a `protected, public, bind(C)` value with the value defined in the C wrapper code.

Classes

C++ classes are transformed to Fortran *derived types*. These types have *type-bound procedures* that mirror the C++ *member functions*. Other SWIG target languages refer to the transformed wrapper classes as “proxy classes” because they act as a proxy to the underlying C++ class.

The Fortran “proxy class” is effectively a C pointer with memory management metadata and type-bound accessors. The C pointer is initialized to `C_NULL_PTR`, and when assigned it can represent a class as a *value* (i.e. the local Fortran code has ownership) or by *reference*. The classes and their implementation are described in detail in the proxy classes section.

Exceptions

By default, a C++ exception will call `std::terminate`, abruptly aborting the Fortran program execution. With the `%exception` feature, C++ exceptions can be caught and handled by the Fortran code by setting and clearing an integer flag. The following snippet from the Examples directory illustrates its use in printing and ignoring an error:

```fortran
use except, only : do_it, ierr, get_serr
call do_it(-3)
if (ierr /= 0) then
  write(0,*) "Got error ", ierr, ": ", get_serr()
  ierr = 0
endif
```

Enabling this exception handling requires `%include`ing a special file and writing a small exception handler.

```
%include <std_except.i>

%exception {
  // Make sure no unhandled exceptions exist before performing a new action
  SWIG_check_unhandled_exception();
  try {
    // Attempt the wrapped function call
    $action
  } catch (std::exception& e) {
    SWIG_exception(SWIG_RuntimeError, e.what() );
  } catch (...) {
    SWIG_exception(SWIG_UnknownError, "An unknown exception occurred");
  }
}

%inline %{
#include <stdexcept>
void do_it(int i)
{
  if (i < 0) throw std::runtime_error("Bad value");
}
%}
```

The above code will wrap (by default) *every* function call. (The standard SWIG `%allowexception` and `%noallowexception` directives can be used to selectively enable or disable exception handling.) Before calling the wrapped function, the call to `SWIG_check_unhandled_exception` ensures that no previous unhandled error exists. If you wish to wrap only a few functions with only specific exceptions, use the "throws" typemap.

When exception handling code is used, SWIG generates a few internal data structures as well as two externally accessible symbols with external C linkage (`ierr` and `get_serr`). Fortran bindings are generated to make the integer and function accessible from the Fortran module.

The names of the integer and string accessor have C linkage and thus must be unique in a compiled program. Since other translation units might have symbols that share the default exception handling names, the user can provide custom names before including the exception handling file:

```
#define SWIG_FORTRAN_ERROR_INT my_ierr
#define SWIG_FORTRAN_ERROR_STR get_my_serr
%include <std_except.i>
```

If you're linking multiple modules together (using %import or otherwise), only one of those modules should define the error integer and accessor by including `<std_except.i>` or `<exception.i>`. Every other module needs to add

```
%include <extern_exception.i>
```

before any other module is `%import`ed (or any other exception-related source files are `%include`d). This inserts the correct exception macros in the wrapper code and *declares* (but does not define) the external-linkage error function and variable. You must also ensure the `SWIG_FORTRAN_ERROR_INT` macro is correctly defined before this include if it's being used upstream.

If you forget to make the above inclusion and an `%import`ed module loads `exception.i`, a SWIG error will be displayed with a reminder of what to do. If *all* of your modules declare `extern_exception.i`, the program will fail to link due to the undefined symbols.

Provided typemaps

There are many ways to make C++ data types interact more cleanly with Fortran types. For example, it's common for C++ interfaces take a `std::string` when they're typically called with string literals: the class can be implicitly constructed from a `const char*` but can also accept a `std::string` if needed. Since Fortran has no implicit constructors, passing a string argument would typically require declaring and instantiating a class for that variable. To mitigate this annoyance, special typemaps are provided that transparently convert between Fortran types and C++ types.

Generally, these typemaps are defined as applying to arguments called `NATIVE`; they can be applied to *all* arguments regardless of name with the `%apply` directive:

```
%apply const char* NATIVE { const char* };
```

or to the output of a specific function such as `const char* get_foo_string(int i);` with

```
%apply const char* NATIVE { get_foo_string };
```

Std::string

A special set of typemaps is provided that transparently converts native Fortran character strings to and from `std::string` classes. It operates essentially like the byte strings described above: it can transparently convert strings of data, even those with embedded null characters, to and from Fortran. This typemap is provided in `<std_string.i>`.

Std::vector

The C `std::vector` class is included with its basic methods. Several typemaps are included alongside it that allow for seamless interoperability with Fortran arrays (with some performance penalty from extra memory allocations and copies). To instantiate wrappers for `std::vector<double>` as a class named `VecDbl`, write

```
%include <std_vector.i>
%template(VecDbl) std::vector<double>;
```

If your code does not use `std::vector` to do any heavy lifting (i.e. your vectors are small and not shared), you can choose to transparently convert those arguments to and from Fortran arrays. This can be done with:

```
%include <std_vector.i>
%template() std::vector<double>;
%apply const std::vector<double> NATIVE& { const std::vector<double>& }
%apply std::vector<double> NATIVE { std::vector<double> }
```

to allow

```
std::vector<double> get_values();
```

to return a Fortran `real(C_DOUBLE), dimension(:), allocatable` array.

Other C++ standard library containers

Other useful types such as `std::map`, `std::set`, have no or minimal implementation. Contributions to these classes (by changes to `swig/Library/fortran/std_{cls}.i`) will be warmly welcomed.

Smart pointers

Like other target languages, SWIG can generate Fortran wrappers to *smart pointers* to C++ objects by modifying the typemaps to that object. A smart pointer is an object whose interface mimics a raw C pointer but encapsulates a more advanced implementation that manages the memory associated with that pointer. Different libraries provide different names and interfaces to smart pointers, but the common `std::shared_ptr` class (and the less common `boost::shared_ptr`) interfaces are provided and can be easily adapted to other similar "smart pointer" types.

When a shared pointer is copied, the pointed-to object is "shared" by the two shared pointer instances, and a reference counter (which keeps track of the number of existing shared pointer instances) is incremented. A shared pointer's reference count is decremented when its destructor is invoked, or if `reset()` is called on the pointer. When the reference count reaches zero, the pointed-to object is deleted.

Wrapping shared pointers with SWIG is as simple as adding the line `%shared_ptr(Foo)` to the source file before the definition of class `Foo` or the wrapping of any function that uses an instance of `Foo`. That macro defines all the necessary typemaps to convert a shared pointer to and from a value, raw pointer, or reference. SWIG does *not* require that all uses of `Foo` be as `shared_ptr<Foo>`: for example, it will correctly dereference the shared pointer when passing it into a function that takes a const reference. Additionally, because shared pointer class supports "null deleters" (i.e. when the reference count reaches zero, the pointed-to data will *not* be deleted), the code can embed a non-owning reference to the data in a shared pointer. In other words, it is OK to return `const Foo&` even when `Foo` is wrapped as a shared pointer.

The following example illustrates the memory management properties of smart pointers. The SWIG interface file is

```
%module spdemo;
%include <std_shared_ptr.i>
%shared_ptr(Foo);

%inline %{
#include <memory>
class Foo {
public:
  explicit Foo(int val) {}
  ~Foo() {}
  const Foo *my_raw_ptr() const { return this; }
};

int use_count(const std::shared_ptr<Foo> *f) {
  if (!f) return 0;
  return f->use_count();
}
%}
```

and the user code is:

```
#define ASSERT(COND) if (.not. (COND)) stop(1)
program main
  implicit none
  use spdemo, only : Foo, use_count
  type(Foo) :: f1, f2

  ASSERT(use_count(f1) == 0)
  f1 = Foo(1) ! Construct
  ASSERT(use_count(f1) == 1)
  f2 = f1 ! Copy shared pointer, not underlying object
  ASSERT(use_count(f1) == 2)
  ASSERT(use_count(f2) == 2)

  f2 = Foo(2) ! Create a new object, assigning the *shared pointer*
                    ! but not replacing the underlying object.
  ASSERT(use_count(f1) == 1)
  ASSERT(use_count(f2) == 1)

  f1 = f2%my_raw_ptr() ! Return a non-shared pointer
                       ! and call the destructor of C++ object 1
  ASSERT(use_count(f2) == 1)

  call f1%release() ! Clear the raw pointer (does not deallocate)
  ASSERT(use_count(f1) == 0)
  call f2%release() ! Destroy the last existing shared pointer
                    ! which then destroys the C++ object 2
  ASSERT(use_count(f2) == 0)

  call f2%release() ! Further calls to release() are allowable null-ops
end program
```

Fortran-to-C array translation

The `<typemaps.i>` library file provides a simple means of passing Fortran arrays by reference. It defines a two-argument typemap `(SWIGTYPE *DATA, size_t SIZE)` that is wrapped as a single Fortran argument, an array of `SWIGTYPE` values. For functions that accept but do not modify an array of values, the signature `(const SWIGTYPE *DATA, size_t SIZE)` is also available.

The following example shows how to apply the typemap to two different functions:

```
%include <typemaps.i>
%apply (SWIGTYPE *DATA, size_t SIZE) { (double *x, int x_length) };
%apply (const SWIGTYPE *DATA, size_t SIZE) { (const int *arr, size_t len) };

void fill_with_zeros(double* x, int x_length);
int accumulate(const int *arr, size_t len);
```

These functions can then be used in Fortran target code:

```
real(C_DOUBLE), dimension(10) :: dbl_values
integer(C_INT), allocatable, dimension(:)  :: int_values
integer(C_INT) :: summed

call fill_with_zeros(dbl_values)
summed = accumulate(int_values)
```

Returning array pointers

The `<view.i>` library file provides an alternate means of converting to and from Fortran array pointers. It translates `std::pair<T*, size_t>` input and output values to and from Fortran array pointers. See the section on pointers and references for cautions on functions returning pointers, but in short, the wrapper code

```
#include <view.i>
```

```
ADD_VIEW(double)
std::pair<double*, size_t> get_array_ptr();
```

is usable in Fortran as

```
real(C_DOUBLE), pointer :: arrptr(:)
arrptr => get_array_ptr()
```

Since this library file is so simple, it can be used as a template for creating transparent wrappers between Fortran arrays and other C++ data types. For example, the following snippet based on `<view.i>` converts a return value of `std::vector<double>& NATIVE` to a Fortran array pointer and applies it to a function `as_array_ptr`.

```
%include <forarray.swg>

// Convert a reference-to-vector return value into a array view.
FORT_ARRAYPTR_TYPEMAP(double, std::vector<double>& NATIVE)
%typemap(out) std::vector<double>& NATIVE %{
  $result.data = $1->empty() ? NULL : $1->data();
  $result.size = $1->size();
%}

%apply std::vector<double>& NATIVE { std::vector<double>& as_array_ptr };
```

MPI compatibility

When wrapping a C++ library that includes MPI support, and the Fortran application uses MPI through the `mpi` module (or `mpif.h` interface), it is often necessary to pass MPI communicators between Fortran and C++. The `<mpi.h>` library header provides bindings to convert between `mpif.h`-defined `integer` communicator values and the standard C `MPI_Comm` datatype. It works by calling the MPI standard functions `MPI_Comm_f2c` and `MPI_Comm_c2f`.

Note that this library inserts include guards into the wrapper code, for the case when it's distributed on a system that doesn't support MPI. It is necessary to inject a configuration file that defines `#define HAVE_MPI` when MPI is available.

This example sets a communicator in C++:

```
%insert("runtime") %{
#include "myconfig.h"
%}
%include <mpi.h>
void set_my_comm(MPI_Comm comm);
```

using the Fortran MPI-native communicator:

```
use mpi
call set_my_comm(MPI_COMM_WORLD)
```

Integer types

One other note to be made about Fortran interoperability concerns the mismatch between default Fortran integers and C++'s `size_type`, which is often used as a function argument. The differing `KIND` of the integers requires that users awkwardly cast values when passing into function calls:

```
call my_vector%resize(INT(n,C_LONG))
```

This nuisance can be simply avoided by replacing occurrences of C's size type with the native Fortran integer type:

```
%apply int { std::size_t }
```

Note of course that if the native integer type is 32-bit and the long type is 64-bit, this will prevent any input larger than `0x7fffffff` from being passed as an argument.

Proxy classes

Each C++ class (with the exception of those wrapped using direct C binding) creates a "proxy class", a unique *derived type* in the Fortran module. Each proxy class holds a single piece of data, a small C-bound struct `SwigClassWrapper`, which contains two simple members: a pointer to C-owned memory, and an enumeration that tracks the ownership of that memory. The proxy class is responsible for tracking ownership of the C++ class and associating that pointer with the corresponding C++ methods.

To introduce the class translation mechanism, we observe the transformation of a simple C++ class

```
class Foo {
public:
  void bar();
};
```

into a Fortran derived type

```
type :: Foo
  type(SwigClassWrapper), public :: swigdata
contains
  procedure :: bar => swigf_Foo_bar
end type
```

The proxy classes that SWIG creates, and how it translates different C++ class features to Fortran, are the topic of this section.

Constructors and Destructors

In C++, the allocation and initialization of a class instance is (almost without exception) performed effectively simultaneously using a constructor. The initialization can be arbitrarily complex, and since the constructor can be overloaded, the instance can be allocated and initialized by several different code paths. In Fortran, initialization can only assign simple scalars and set pointers to null.

However, "construction" can be done separately by an unbound procedure, which uses an `interface` to share the name of the class:

```
type(Foo) :: f
type(Foo) :: g
f = Foo()
g = Foo(123)
call f%do_something()
call g%do_something_else()
```

Even though the Fortran 2003 standard specifies when local variables become *undefined* (and are *finalized* if they have a `FINAL` subroutine), support for finalization in many compilers still in active use is not entirely reliable. Rather than relying on the finalization mechanics to clean up and free a C++ object, destructors for the C++ wrappers wrapped as a `release` procedure:

```
call f%release()
call g%release()
```

To avoid leaking memory, `release` should *always* be called when the proxy class instance is no longer needed. It will free memory if appropriate and reset the C pointer to `NULL`. Calling `release` on an uninitialized variable (or a variable that has been released) is a null-op.

Because Fortran 2003 does specify support for a special `FINAL` procedure to clean up local or dynamic variables, the call to `release()` can be replaced by adding a `FINAL` procedure. The SWIG Fortran interface can generate this procedure, which will call the C++ destructor:

```
%feature("final") Foo;
%include "Foo.h"
```

However, this feature is relatively untested and its behavior could be compiler-dependent, so extreme caution is recommended when enabling it.

Member functions

SWIG generates unique, private procedure names (generally `swigf_{classname}_{funcname}` for each class and function. These procedures are bound to the type. If function overloading is used, "generic" procedures will be added to the derived type.

Type-bound procedures in Fortran proxy classes are treated exactly the same as for native derived types:

```fortran
integer(C_INT) :: value
type(Foo) :: food
food = Foo()
call food%do_something()
value = food%get_something()
```

Function overloading for derived types is implemented using *generic interfaces*. Each overloaded function gets a unique internal symname, and they are bound together in a generic interface. For example, if a member function `doit` of class `Action` is overloaded, a generic binding will be generated inside the Fortran proxy derived type:

```fortran
  procedure, private :: doit__SWIG_0 => swigf_Action_doit__SWIG_0
  procedure, private :: doit__SWIG_1 => swigf_Action_doit__SWIG_1
  generic :: doit => doit__SWIG_0, doit__SWIG_1
```

As with free functions, a member function returning `void` cannot be overloaded with a function returning non-void.

Member data

SWIG generates member functions for class member data in the same way that it generates free functions for global variables. Each public member produces a "getter", and unless the data is marked `const`, it generates a "setter".

For a struct

```c
struct Foo {
  int val;
};
```

the interface to an instance and its data is:

```fortran
type(Foo) :: f
f = Foo()
call food%set_val(123)
value = food%get_val()
```

Inheritance

Single inheritance in C++ is mirrored by Fortran using the `EXTENDS` attribute. For classes with virtual methods, the user should keep in mind that function calls are dispatched through C++. In other words, even if you call a base-class member function in Fortran that wraps a derived class instance, the correct virtual function call will be dispatched.

Fortran has no mechanism for multiple inheritance, so this SWIG target language does not support it. The first base class listed that has not been `%ignore`d will be treated as the single parent class.

There is no intrinsic way to `dynamic_cast` to a daughter class, but if a particular casting operation is needed a small inline function can be created that should suffice:

```
%inline %{
Derived& base_to_derived(Base& b) {
    return dynamic_cast<Derived&>(b);
}
%}
```

(Note that this function will *not* transfer ownership to the new object. Doing that is outside the scope of this chapter.)

The implementation of function overloading in the Fortran types can cause compiler errors when member functions are *shadowed* or *overridden* in a daughter class. First, Fortran requires essentially that overriding procedures must have the exact same function signature *including the names of the dummy arguments*. Overriding functions in C++ merely require the same parameter types. Second, Fortran does not allow a procedure in a parent type to be “shadowed” by the extending type as C++ does. Finally, a non-generic procedure in the parent type cannot be shadowed by a generic procedure.

Memory management

A single Fortran proxy class must be able to act as a value, a pointer, or a reference to a C++ class instance. When stored as a value, a method must be put in place to deallocate the associated memory; if the instance is a reference, that same method cannot double-delete the associated memory. Finally, C++ functions must be able to send Fortran pointers both *with and without* owning the associated memory, depending on the function. Finally, assignment between Fortran classes must preserve memory association.

Fortran’s “dummy argument” for the return result of any function (including generic assignment) is `intent(out)`, preventing the previous contents (if any) of the assignee from being modified or deallocated. At the same time, the assignment operator must behave correctly in both of these assignments, which are treated identically by the language:

```fortran
type(Foo) :: a, b
a = make_foo()
b = a
```

Note that unlike Python, `b` is not a pointer to `a`; and unlike C++, `b` is not copy-constructed from `a`. Instead, `a` is assigned to `b` using the `assignment(=)` operator. Likewise, `a` is not "constructed" on the second line: there is no return value optimization as in C++. Instead, `make_foo` returns a temporary `Foo`, and that *temporary* is assigned to `a`.

Because these two assignments are treated equally and a temporary is created in only one of them, we have to be clever to avoid leaking or double-deleting memory.

Ideally, as was done in Rouson's implementation of Fortran shared pointers, we could rely on the `FINAL` operator defined by Fortran 2003 to release the temporary's memory. Unfortunately, even 15 years after the standard was ratified, support for `FINAL` is patchy and unreliable.

Our solution to this limitation is to have the `Foo` proxy class store not only a pointer to the C data but also a state enumeration `self%swigdata%mem` that describes memory ownership. The enumeration needs to have at least three options:

- The memory is *owned* by the proxy class (and must be deleted when calling `release()`);
- The proxy class is a *reference* to memory owned by C/C++ (returned by either a raw pointer or a reference);
- The memory is being allocated and returned from a function, but it must be captured by the left hand side.

This last option is roughly analogous to the behavior of the deprecated `std::auto_ptr`, which was the predecessor to C++11's `move` semantics. Besides the above flags, we also define an uninitialized state `NULL` for convenience, and a "const reference" state to enable const correctness. These flags are set by the SWIG `out` typemaps in the C wrapper code: if memory is being allocated, the return flag is `MOVE`; if a pointer is being returned, `REF` (or `CREF` in the const case) is used.

The crucial trick is to implement an assignment operator that correctly copies, allocates, or moves memory based on the flags on the left- and right-hand sides, and sets a new memory state on the recipient. By resetting the state flag in a generic assignment operator, we guarantee that *only* temporary classes will ever have the `MOVE` state.

For the operation `self = other`, where `other` may be a return value from a function or an existing object, a variety of slightly different actions are taken depending on the memory ownership flags of `self` and `other`. The following table describes the C++ action performed, where `pself` is the pointer being managed by `self`, `pother` is the pointer managed by `other`, both classes are of type `This`, and `std::move` is used in the local namespace. For brevity, the `SWIG_` prefix to the memory flags has been omitted.

Self	Other	*Action*
NULL	NULL	(none)
NULL	MOVE	`pself = pother;`
NULL	OWN	`pself = new This(pother);`
NULL	REF/CREF	`pself = pother;`
OWN	NULL	`delete pself; pself = NULL;`
OWN	MOVE	`*pself = move(*pother); delete pother;`
OWN	OWN	`*pself = *pother;`
OWN	REF/CREF	`*pself = *pother;`
REF	NULL	`pself = NULL;`
REF	MOVE	`*pself = move(*pother); delete pother;`
REF	OWN	`*pself = *pother;`
REF	REF/CREF	`*pself = *pother;`
CREF	NULL	`pself = NULL;`
CREF	MOVE	(error)
CREF	OWN	(error)
CREF	REF/CREF	(error)

The above operations are designed to preserve C++ semantics: if an proxy object owning memory is assigned, then any existing objects pointing to that memory will reflect the newly assigned value. This is important for classes whose interface relies on returning mutable references.

The fact that some classes disallow combinations of copy/move constructors and assignment complicates the task of evaluating the above actions. SWIG's built-in parsing of class features will detect what constructors (if any) are available, whether an assignment operator is defined, and whether the destructor is public. If C++11 is enabled in the compiler that uses the SWIG code, standard library type traits override the SWIG-parsed features and additionally enable move construction and move assignment. For more complicated cases – such as classes with default assignment operators but with `const` data members – you may define a traits class that explicitly specifies the allowable operations. This may be needed to avoid compiler errors; an example is in `<std_pair.i>`.

In all cases, unless the `final` feature is enabled (and works with the Fortran compiler), `release` should be called on every proxy class instance.

Opaque class types

SWIG's default Fortran type (the `ftype` typemap) for generic types such as classes (`SWIGTYPE`) is:

```
%typemap(ftype) SWIGTYPE "type($fclassname)"
```

The special symbol `$fclassname` is replaced by the symbolic name of the class that matches the typemap. For example, if `std::vector<double>` is instantiated:

```
%template(Vec_Dbl) std::vector<double>;
```

then `Vec_Dbl`, the name of the derived type, will replace `$fclassname`.

If a class has *not* been wrapped but is encountered (e.g. in a function argument or return value), a warning will be emitted: no Fortran derived type has been generated to correspond to the C++ class. A new derived type `SwigUnknownClass` will be generated that simply holds an opaque pointer to the C++ object. Currently the `SwigUnknownClass` is private to each module and thus cannot be used by code outside the module. Conceivably it could be made public so that it can be used by the C interface as a raw `void*` pointer.

Proxy class wrapper code

The Fortran wrapper code generated for each function can be extended in multiple ways besides using the `fin` and `fout` typemaps. A specific function can have code prepended to it using the `%fortranprepend` macro, which is a compiler macro for `%feature("fortran:prepend")`, and appended using `%fortranappend`, which aliases `%feature("fortran:append")`.

For advanced cases, the function or subroutine invocation can be embedded in another layer of wrapping using the `%feature("shadow")` macro. The special symbol `$action` will be replaced with the usual invocation.

Advanced details and usage

This section describes some of the advanced features that underpin the SWIG Fortran wrapping. These features allow extensive customization of the generated C/Fortran interface code and behavior.

Typemaps

SWIG Fortran extends the typemap system of SWIG with additional typemaps, modeled after the Java target language's typemaps. They provide for translating

C++ data to and from an ISO-C compatible datatype, and from that datatype to native Fortran types. These special typemaps are critical to understanding how SWIG passes data between Fortran and C++.

ISO C Wrapper interface

SWIG-generated Fortran code works by translating C++ data types to simple C types compatible with ISO C binding, then translating the data types to more complex Fortran data types. The C-compatible types are known as the "intermediate layer".

SWIG Fortran defines two new typemaps to declare the data types used by Fortran and C in the intermediate layer, and two typemaps for translating the intermediate layer types to and from the final Fortran types.

To pass Fortran-2003 compatible `BIND("C")` or `ISO_C_BINDING` types between C++ and Fortran, you must declare a compatible `ctype` and `imtype`. The `ctype` is the C datatype used by the wrapper and intermediate layer, and `imtype` is the equivalent Fortran datatype. These datatypes generally must be either fundamental types or structs of fundamental types. For example, as described in the Fundamental types section, the `int` C type is compatible with `integer(C_INT)` Fortran type. However, because Fortran prefers to pass data as pointers, SWIG defines `int*` as the `ctype` for `int`. Otherwise the `imtype` would have to be `integer(C_INT), value`.

The `ctype` and `imtype` each have keywords that are usually required. By default, `ctype` corresponds to an *input* value, i.e. a function argument. Often the *output* value of a function is a different type (e.g. simply an `int` instead of `int*`). The `out` keyword allows this to be overridden:

```
%typemap(ctype, out="int") int
  "const int*"
```

The `imtype` is used both as a dummy argument *and* as a temporary variable in the fortran conversion code. Because these also may have different signatures, an `in` keyword allows the dummy argument to differ from the temporary:

```
%typemap(imtype, in="integer(C_INT), intent(in)") int
  "integer(C_INT)"
```

Fortran proxy datatype translation

The `fin` and `fout` typemaps are Fortran proxy wrapper code analogous to the `in` and `out` in the C wrapper code: they are used for translating native Fortran objects and types into types that can be transmitted through the ISO C intermediate code. For example, to pass a class by reference, the Fortran class `class(SimpleClass) :: self` is converted to the corresponding C class via the stored C pointer using the `fin` typemap, which is expanded to:

```
farg1 = self%swigdata%ptr
```

This argument is then passed into the C function call:

```
fresult = swigc_make_class(farg1)
```

and the output is translated back via the `fout` typemap, which in this case expands to:

```
swig_result%swigdata%ptr = fresult
```

Allocating local Fortran variables in wrapper codes

Advanced SWIG users may know that

```
%typemap(in) int (double tempval) { /.../ }
```

is a way to declare a temporary variable `tempval` in the C wrapper code. The same feature is emulated in the special typemaps `findecl` and `foutdecl`, which are inserted into the variable declaration blocks when the corresponding types are used. If `findecl` allocates a temporary variable, the `ffreearg` typemap (analogous to the `freearg` typemap for C `in` arguments) can be used to deallocate it.

An example for returning a native `allocatable` Fortran string from a C++ string reference must declare a temporary array pointer to the C data, then copy the result into a Fortran string.

```
%typemap(ftype, out="character(kind=C_CHAR, len=:), allocatable")
    const std::string&
"character(kind=C_CHAR, len=*), target"

// Fortran proxy translation code: temporary variables for output
%typemap(foutdecl) const std::string&
%{
 integer(kind=C_SIZE_T) :: $1_i
 character(kind=C_CHAR), dimension(:), pointer :: $1_chars
%}

// Fortran proxy translation code: convert from imtype $1 to ftype $result
%typemap(fout) const std::string&
%{
  call c_f_pointer($1%data, $1_chars, [$1%size])
  allocate(character(kind=C_CHAR, len=$1%size) :: $result)
  do $1_i=1,$1%size
    $result($1_i:$1_i) = $1_chars($1_i)
  enddo
%}
```

Special class typemaps

To facility the wrapping and customizability of C++ classes, there are a few additional special typemaps that only apply to classes. They generally should not need to be modified.

The `fdata` typemap declares the data object that is stored by the Fortran proxy class. Note that only the base class of any inheritance hierarchy contains this data.

The `fdestructor` typemap becomes the Fortran wrapper code for the `release` type-bound procedure. The special token `$action` is replaced by the call to the C wrapper for the destructor. Currently, all classes have the same destructor action but this may change.

Fragments

The `%insert(section) %{ ...code... %}` directive can be used to inject code directly into the C/C++ wrapper file as well as the Fortran module file. The Fortran module uses several additional sections that can be used to insert arbitrary extensions to the module. For example, if an `%insert` directive is embedded within a class `%extend`, new type-bound procedures can be manually added to the derived type.

The generated C++ wrapper file has the following sections denoted by `{sectionname}`

```
{begin}
{runtime}
{header}
#ifdef __cplusplus
extern "C" {
#endif
{wrapper}
#ifdef __cplusplus
}
#endif
{init}
```

The generated Fortran module looks like:

```
{fbegin}
module [MODULE_NAME]
 use, intrinsic :: ISO_C_BINDING
 {fmodule}
 implicit none
 private
 {fpublic}
 ! module generic interfaces
 {fparams}
 {ftypes}
interface
 {finterfaces}
end interface
contains
 {fwrapper}
end module
```

Direct C binding

It is sometimes desirable to simply expose C functions and types to Fortran. (Of course, this may be done only when the data types involved are ISO-C compatible.)

Generating C-bound Fortran types from C structs

In certain circumstances, C++ structs can be wrapped natively as Fortran `BIND(C)` derived types, so that the underlying data can be shared between C and Fortran without any wrapping needed. Structs that are "standard layout" in C++ can use the `%bindc` feature to translate

```
struct BasicStruct {
  int foo;
  double bar;
};
```

to

```
type, bind(C) :: BasicStruct
  integer(C_INT), public :: foo
  real(C_DOUBLE), public :: bar
end type
```

Roughly speaking, standard layout structs have no virtual member functions, inheritance, or C++-like member data. All structs in C are compatible with Fortran, unless they bit have fields or use the C99 feature of "flexible array members".

Currently the C binding feature for structs must be activated using a special macro `%fortran_bindc_struct`:

```
%fortran_bindc_struct(BasicStruct);
```

In C++, these structs must be "standard layout", i.e. compatible with C.

Calling `%fortran_bindc_struct(Foo)` inhibits default constructor/destructor generation for the class, and it sets up the necessary type definitions to treat the struct as a fundamental type.

Every member of the struct must be `BIND(C)` compatible. This is enforced with a separate typemap `bindc` that translates the member data to Fortran type members. For example, the basic `int` mappings are defined (using macros) as:

```
%typemap(bindc) int      "integer(C_INT)"
%typemap(bindc) int*     "type(C_PTR)"
%typemap(bindc) int[ANY] "integer(C_INT), dimension($1_dim0)"
%typemap(bindc) int[] = int*;
```

The `bindc` typemap is used when wrapping global constants as well.

Interfaces with Fortran C-bound types

If types defined in the SWIG Fortran module are to be used as part of the interface (as is the case with structs), it is necessary to "import" the type into the interface to use it. This is accomplished by the `import` keyword argument to the `imtype` typemap. For example, whenever the following typemap is used in the intermediate wrapper:

```
%typemap(imtype, import="SwigArrayWrapper")  FooArray
  "type(SwigArrayWrapper)";
```

an `import` directive will be inserted into the Fortran proxy function:

```
module thinvec
 use, intrinsic :: ISO_C_BINDING
 implicit none

 type, public, bind(C) :: SwigArrayWrapper
   type(C_PTR), public :: data
   integer(C_SIZE_T), public :: size
 end type
 interface
 subroutine swigc_foo(farg1) &
   bind(C, name="swigc_foo")
   use, intrinsic :: ISO_C_BINDING
   import :: SwigArrayWrapper     ! Will not compile without this line
   type(SwigArrayWrapper) :: farg1
 end subroutine
```

This extra typemap trickery should only be needed if you're generating bound types without using the `%fortran_bindc_struct` macro.

Generating direct Fortran interfaces to C functions

In addition to generating functions with translation code, it is also possible to specify that a function be directly *bound* and not *wrapped*. For this feature to work correctly, all function arguments and return types must be inherently Fortran/C interoperable. If using C++, the function must be defined using `extern "C"` linkage; and in fact, when SWIG is asked to wrap a function with that linkage, it defaults to binding it. Use the `%nobindc my_func_name;` feature to suppress this behavior.

The C++ code:

```
extern "C" {
// These functions are simply bound, not wrapped.
void print_sphere(const double origin[3], const double* radius);
}
```

is automatically translated into

```
subroutine print_sphere(origin, radius) &
    bind(C, name="print_sphere")
  use, intrinsic :: ISO_C_BINDING
  real(C_DOUBLE), dimension(3), intent(in) :: origin
  real(C_DOUBLE), intent(in) :: radius
end subroutine
```

Known Issues

A number of known limitations to the SWIG Fortran module are tracked on GitHub.