Automated Fortran–C++ Bindings for Large-Scale Scientific Applications

Seth R Johnson
HPC Methods for Nuclear Applications Group
Nuclear Energy and Fuel Cycle Division
Oak Ridge National Laboratory

github.com/swig-fortran
Overview

• Introduction
• Tools
• SWIG+Fortran
• Strategies
• Example libraries
Introduction
How did I get involved?

• SCALE (1969–present): Fortran/C++
• VERA: multiphysics, C++/Fortran
• MPACT: hand-wrapped calls to C++ Trilinos
Project background

• Exascale Computing Project: at inception, many scientific app codes were primarily Fortran

• Numerical/scientific libraries are primarily C/C++

• Expose Trilinos solver library to Fortran app developers: ForTrilinos product
ECP: more exascale, less Fortran

ECP application codes over time (credit: Tom Evans)
Motivation

• C++ library developers: expand user base, more opportunities for development and follow-on funding
• Fortran scientific app developers: use newly exposed algorithms and tools for your code
• Multiphysics project integration: in-memory coupling of C++ physics code to Fortran physics code
• Transitioning application teams: bite-size migration from Fortran to C++
Tools
Wrapper “report card”

- **Portability**: Does it use standardized interoperability?
- **Reusability**: How much manual duplication needed for new interfaces?
- **Capability**: Does the Fortran interface have parity with the C++?
- **Maintainability**: Do changes to the C++ code automatically update the Fortran interface?
- **Robustness**: Is it possible for silent failures to creep in?
- **Integration**: How much overhead is needed to get the glue code working in development/deployment?
Hand-rolled binding code

• Often based on hand-rolled C interface layer
• Often targets F77/90 using configure-time bindings
• Examples: SuperLU, STRUMPACK, TASMANIAN

<table>
<thead>
<tr>
<th>Portability</th>
<th>😞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>😳</td>
</tr>
<tr>
<td>Capability</td>
<td>😐</td>
</tr>
<tr>
<td>Maintainability</td>
<td>😞</td>
</tr>
<tr>
<td>Robustness</td>
<td>😨</td>
</tr>
<tr>
<td>Integration</td>
<td>😒/🙂</td>
</tr>
</tbody>
</table>
Project-specific scripts

- Text processing engines hardcoded to a particular project’s data types, header formatting
- Simple translations of function signatures and types to “glue code”
- Examples: MPICH, HDF5, Silo, PETSc

<table>
<thead>
<tr>
<th>Portability</th>
<th>😞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>😐</td>
</tr>
<tr>
<td>Capability</td>
<td>😐</td>
</tr>
<tr>
<td>Maintainability</td>
<td>😐</td>
</tr>
<tr>
<td>Robustness</td>
<td>😊</td>
</tr>
<tr>
<td>Integration</td>
<td>😞/😊</td>
</tr>
</tbody>
</table>
Automated code generators (manual C++ declaration)

- **ClX**: in-house ORNL tool (template substitution)
- **Shroud**: recent LLNL development (custom YAML)

<table>
<thead>
<tr>
<th>Portability</th>
<th>😊</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>😊</td>
</tr>
<tr>
<td>Capability</td>
<td>😊</td>
</tr>
<tr>
<td>Maintainability</td>
<td>😓</td>
</tr>
<tr>
<td>Robustness</td>
<td>😊</td>
</tr>
<tr>
<td>Integration</td>
<td>😐/😊</td>
</tr>
</tbody>
</table>
Automated code generators (integrated C++ parsing)

- SWIG+Fortran: fork of Simplified Wrapper Interface Generator
SWIG+Fortran
SWIG: Simplified Wrapper and Interface Generator

- Generate *interfaces* to existing C and C++ code and data types so that a *target language* can invoke functions and use the data
- “Glue” code: flat C-linkage wrappers to C++ functions, corresponding interfaces in target language
- Does *not* couple target languages to other target languages
- Does *not* parse target languages or create C++ proxy wrappers

Supported Languages

- Allegro CL
- C#
- CFFI
- CLISP
- Chicken
- D
- Go
- Guile
- Java
- Javascript
- Lua
- Modula-3
- Mzscheme
- OCAML
- Octave
- Perl
- PHP
- Python
- R
- Ruby
- Scilab
- Tcl
- UFFI
SWIG execution sequence

• SWIG reads `.i` interface file which may `%include` other C/C++ headers to parse them
  • Interface file can be as little as a couple of lines for simple interfaces
  • More difficult C++/Fortran conversions require more interface code

• SWIG *generates source code files*: `.cxx` and `.f90`
  • This wrapper code can be distributed like regular source files
  • Library users *don’t need to know SWIG*
Control flow and data conversion

- Fortran 2003 standard defines C-compatible datatypes
- Use only Fortran-compatible ISO C datatypes
- Minimize data movement of numerical arrays
Features

- **Primitive types**
- Enumerations
- **Classes** (with inheritance)
- C strings and std::string
- Function pointers
- **Arrays** and std::vector

- Function overloading
- **Template instantiation**
- Compile-time constants
- **Exception handling**
- OpenACC and Thrust
Addition with “native” int: input and generated C code

```c
#include "simplest.h"

SWIGEXPORT int _wrap_add(int const *farg1, int const *farg2) {
    int fresult;
    int arg1;
    int arg2;
    int result;
    arg1 = (int)(*farg1);
    arg2 = (int)(*farg2);
    result = add(arg1, arg2);
    fresult = (int)(result);
    return fresult;
}
```

**simplest_wrap.c**

**F/C interface**

**Input argument conversion**

**Wrapped function call**

**Output argument conversion**

**simplest.h**

```c
#ifndef simplest_h
#define simplest_h

int add(int a, int b);
#endif

simplest.h
```

**simplest.i**

```c
%module simplest

%typemap(ftype, in="integer, intent(in)"
    int "integer"
%typemap(fin) int "$1 = int($input, C_INT)"
%typemap(fout) int "$result = int($1)"
%include "simplest.h"
```

**simplest.i**
Addition with “native” int: generated Fortran

```fortran
module simplest
use, intrinsic :: ISO_C_BINDING
implicit none
private
public :: add
interface
  function swigc_add(farg1, farg2) &
    bind(C, name="_wrap_add") &
    result(fresult)
    integer(C_INT), intent(in) :: farg1
    integer(C_INT), intent(in) :: farg2
    integer(C_INT) :: fresult
  end function
end interface
contains
....
end module

simplest.f90
```
Simple addition function: the part app devs care about

```c
SWIGEXPORT int swigc_add(int const *farg1, int const *farg2);
```

```fortran
module simplest
  use, intrinsic :: ISO_C_BINDING
  ...
  contains
  function add(a, b) 
    result(swig_result)
    integer :: swig_result
    integer, intent(in) :: a
    integer, intent(in) :: b
    ...
  end function
end module
```

```fortran
program main
  use simplest, only : add
  write (0,*) add(10, 20)
end program
```

```bash
$ ./main.exe
30
```
Automatic **BIND(C)** wrapping

```
%module bindc
  %fortranbindc;
  %fortran_struct(Point)

%inline {
  typedef struct { float x, y, z; } Point;
  void print_point(const Point* p);
  void make_point(Point* pt, const float xyz[3]);
}

module bindc
  ...
  type, bind(C), public :: Point
    real(C_FLOAT), public :: x
    real(C_FLOAT), public :: y
    real(C_FLOAT), public :: z
  end type Point
...
interface
  subroutine print_point(p) &
    bind(C, name="print_point")
  use, intrinsic :: ISO_C_BINDING
  import :: point
  type(Point), intent(in) :: p
  end subroutine
  subroutine make_point(pt, xyz) &
    bind(C, name="make_point")
  use, intrinsic :: ISO_C_BINDING
  import :: point
  type(Point) :: pt
  real(C_FLOAT), dimension(3), intent(in) :: xyz
  end subroutine
end interface
end module
```
**Templated class**

```cpp
// Insert raw C++ code into generated wrapper file
%module "templated"
{
#include "templated.hpp"
%
%include "templated.hpp"

// Instantiate templated classes
%template (Thing_Int) Thing<int>
%template (Thing_Dbl) Thing<double>

// Instantiate and overload a function
%template (print_thing) print_thing<int>
%template (print_thing) print_thing<double>

// Templated class

template<typename T>
class Thing {
    T val_; 
    public:
        Thing(T val);
        T get() const;
};

template<typename T>
void print_thing(const Thing<T>& t);
```
Templated class: generated Fortran wrapper code

```fortran
module templated
    ! class Thing< int >
    type, bind(C) :: SwigClassWrapper
    type(C_PTR), public :: cptr = C_NULL_PTR
    integer(C_INT), public :: cmemflags = 0
end type

    ! class Thing< double >
    type, public :: Thing_Dbl
    ... end type Thing_Dbl
end module
```

### Memory ownership

- `integer, parameter, public :: swig_cmem_own_bit = 0`
- `integer, parameter, public :: swig_cmem_rvalue_bit = 1`

### Opaque class wrapper

- `type, bind(C) :: SwigClassWrapper`
- `type(C_PTR), public :: cptr = C_NULL_PTR`
- `integer(C_INT), public :: cmemflags = 0`

### Overloaded function

- Call delete if we “own” the memory

### Second template instantiation

- `interface print_thin
  module procedure swigf_print_thin__SWIG_1,
  swigf_print_thin__SWIG_2
end interface`

- `public :: print_thin`

```fortran
interface print_thin
  module procedure swigf_print_thin__SWIG_1,
  swigf_print_thin__SWIG_2
end interface
```

### Fortran proxy class

- `use, intrinsic :: ISO_C_BINDING`
- `class(Thing_Int), intent(inout) :: self`
- `type(SwigClassWrapper) :: farg1`
- `farg1 = self%swigdata`
- `if (btest(farg1%cmemflags, swig_cmem_own_bit)) then`
  - `call swigc_delete_Thing_Int(farg1)`
- `endif`
- `farg1%cptr = C_NULL_PTR`
- `farg1%cmemflags = 0`
- `self%swigdata = farg1`

```fortran
contains
  subroutine swigf_release_Thing_Int(self)
    use, intrinsic :: ISO_C_BINDING
    class(Thing_Int), intent(inout) :: self
    type(SwigClassWrapper) :: farg1
    farg1 = self%swigdata
    if (btest(farg1%cmemflags, swig_cmem_own_bit)) then
      call swigc_delete_Thing_Int(farg1)
    endif
    farg1%cptr = C_NULL_PTR
    farg1%cmemflags = 0
    self%swigdata = farg1
  end subroutine
end interface
```

```fortran
end module
```

 templated.f90 (2/2)
%module except
%include <std_except.i>
%exception {
    SWIG_check_unhandled_exception();
    try {
        $action
    } catch (const std::exception& e) {
        SWIG_exception(SWIG_RuntimeError, e.what());
    }
}
%
#include <stdexcept>
#include <iostream>
%
%inline {
    void do_it(int i) {
        if (i < 0)
            throw std::logic_error("NOOOOO");
        std::cout << "Yes! I got " << i
            << std::endl;
    }
    void do_it_again(int i) { do_it(i); }
}
Array views

```plaintext
%module algorithm
{
#include <algorithm>
%

#include <typemaps.i>
%apply (SWIGTYPE *DATA, size_t SIZE) {
  (int* data, std::size_t size) }

%inline {
void sort(int* data, std::size_t size)
{
  std::sort(data, data + size);
}
}
```
Performance considerations

- Small overhead for each wrapped function call
- Link-time optimization can mitigate
- Test problem: toy numerical library with CRS matrix class
- Results: sparse matrix–vector multiply for 3000×3000 Laplacian (average of 40 runs)

Data credit: Andrey Prokopenko
Strategies
How to add new capabilities that “should be” library functions to your Fortran app

- Wait for standards committee… then wait for compiler implementors
- Roll your own (e.g. numerical recipes)
- Fortran package ecosystem/fortran (FPM)*
- Contribute Fortran interfaces to C++ libraries
- Transition toward C++ by writing new C++ classes and using SWIG to couple to existing Fortran codebase
Adding Fortran interfaces to your C++ library

• Expose low-level C++ objects: Fortran user code looks like C++
• Write thin high-level C++ wrappers for targeted capabilities
• Target idiomatic usage for Fortran apps
Idiomatic Fortran

- Accept arrays rather than scalars or iterators as function inputs
- Return “status values” as optional argument of subroutine rather than result value of function
- Indexing convention: first element of an array is 1, possibly use 0 to indicate “not found”
- Use native Fortran types where possible rather than C types

SWIG+Fortran assists with all the above
Challenges

• Tension between Fortran/C “static” nature and increasingly dynamic C++ capabilities
  • Kokkos relies on lambdas and compile-time detection of backend
  • Numerous C++11 libraries are interface-only with extensive templating, auto keyword, initializer lists, etc.
  • Explicit compile-time interface needed to bind C++ and Fortran
• SWIG-generated interface must be configuration-independent
Example libraries
Flibcpp: Fortran bindings for C++ standard library

- Speed and reliability of C++ standard library
- Trivial installation and downstream usage
- App development requires only idiomatic Fortran
- Sort/search/set, sets, vectors, strings, PRNGs, ...
- Callbacks for sorting too!

```fortran
use flc_algorithm, only : argsort
implicit none
integer, dimension(5) :: iarr = [2, 5, -2, 3, -10000]
integer(C_INT), dimension(5) :: idx

call argsort(iarr, idx)
! This line prints a sorted array:
write(*,*)(iarr(idx))

use flc_random, only : Engine, normal_distribution
real(C_DOUBLE), dimension(20) :: arr
type(Engine) :: rng

rng = Engine()
call normal_distribution(8.0d0, 2.0d0, rng, arr)
```

https://github.com/swig-fortran/flibcpp
ForTrilinos

• Includes low-level Tpetra/Teuchos objects and high-level solver interfaces

• Inversion-of-control for Fortran implementations of operators

ForTrilinos coauthor: Andrey Prokopenko

```module myoperators
use forteuchos
use fortpetra
implicit none

type, extends(ForTpetraOperator) &
:: TriDiagOperator

type(TpetraMap) :: row_map, col_map, &
domain_map, range_map

contains

procedure :: apply => my_apply
procedure :: getDomainMap &
=> my_getDomainMap
procedure :: getRangeMap &
=> my_getRangeMap
procedure :: release &
=> delete_TriDiagOperator

end type

interface TriDiagOperator
procedure new_TriDiagOperator
end interface

! ...
end module```
program test_thrustacc
use flhpc, only :: sort
implicit none
integer, parameter :: n = 32768
integer :: i
real, dimension(:), allocatable :: a

! Generate N uniform numbers on [0,1)
allocate(a(n))
call random_number(a)

!$acc data copy(a)
!$acc kernels
do i = 1,n
  a(i) = a(i) * 10 + i
end do
!$acc end kernels
call sort(a)
!$acc end data
write(*,*) sum(a)
end program

Flibhpc: Thrust/OpenACC/MPI

• Uses SWIG to integrate acc_deviceptr/C_DEVPTR with Thrust device_ptr<T> to pass

• MPI integration (convert F77/90-style MPI integers into C MPI_Comm objects)

Sneak peek
Other ECP codes using SWIG+Fortran

• SUNDIALS
• TASMANIAN
• DTK
• STRUMPACK
• SCALE
Summary

• Exascale era is another driver for inter-language operability
• Coupling can be driven by apps and/or libraries
• SWIG+Fortran produces robust, idiomatic glue code
• New library bindings can give a taste of C++ capabilities to Fortran codes

<table>
<thead>
<tr>
<th></th>
<th>Github</th>
<th>Spack</th>
<th>E4S</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWIG+Fortran</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ForTrilinos</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flibcpp</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Flibhpc</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
https://github.com/swig-fortran

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.