



EXPOSING FORTRAN DERIVED TYPES TO C AND OTHER LANGUAGES

By Alexander Pletzer, Douglas McCune, Stefan Muszala, Srinath Vadlamani, and Scott Kruger

When building large scientific codes, you might have to mix different programming languages. The authors show how to bridge the interoperability gap between Fortran 90/95 and C, and from C to other languages, with working code examples.

In the past decade, scientific codes written in Fortran have increasingly come to rely on derived types, also known as compound or record types, for multiple reasons. Using derived types can simplify the API and let the code evolve without breaking backward compatibility. Adding new features might involve merely extending the derived types or attaching new methods (the existing methods' signatures needn't change). Although derived types also provide a mechanism for data hiding and encapsulation, perhaps more importantly, they're a step toward object-oriented programming whereby a composite object is created, has a life span during which it can be altered, and is finally destroyed. The combination of data members and procedures acting on the derived type form a "class" in the same sense as the `class` keyword in C++, Java, and Python, so programmers can emulate many object-oriented constructs in languages such as Fortran 90/95 (or C, for that matter) that don't formally have an object-oriented syntax.

At Tech-X, a company specializing in object-oriented scientific programming on parallel architectures, our daily tasks include integrating scientific legacy codes into software infrastructures. One example of such infrastructure is the Framework Application for Core-Edge Transport Simulation (<https://facets.txcorp.com/facets>), a multiphysics effort led by Tech-X to bring whole-device modeling capability to the fusion community. For either performance or for historical reasons, the legacy codes we encounter are typically written in Fortran, with infrastructure code written in C++ for flexibility. In the process of integrating these codes, we constantly need to call Fortran from C++. To address mixed language programming issues, Tech-X is developing a new Fortran 2003 binding for Babel¹ that focuses on Fortran derived types and C/C++ `structs`. Babel currently provides language interoperability between C, C++, FORTRAN 77, Fortran 95, Java, and Python, and lets these languages operate in a single address space;² it also provides a programming-lan-

guage-neutral specification of interfaces via the Scientific Interface Definition Language.

Although we're aware of other projects that mix Fortran 90 and C++,³ no comprehensive, general, or widely known methodology appears to address the issue of exposing Fortran 90 derived types to other languages. We aim to correct this situation by presenting three strategies here (along with working code examples). Our basic assumption is that C acts as a lingua franca for interlanguage communication, so solving the C–Fortran 90 interoperability problem also solves the Fortran–X interoperability problem, where X is any language with a C-callable interface (such as Python, Ruby, or Java). The methodology presented here is compatible with existing tools for automatically generating cross-language bindings, including Babel, f2py (<http://cens.ioc.ee/projects/f2py2e>), and SWIG (www.swig.org). Babel, f2py, and SWIG offer a similar functionality, but none of them directly support interfaces that contain Fortran derived types, although Babel and f2py plan to do so in the future.

Why Interoperability Looks Hard

FORTAN 77 was never officially interoperable with C, although rules have been developed over the years to let C user code invoke Fortran code and vice versa in a portable way. One such rule is that a C `int` maps to a Fortran `integer`, a C `float` to a Fortran `real`, and so forth. Another rule handles the way Fortran compilers mangle procedure names, typically by attaching an underscore. A third rule assumes that Fortran procedure arguments are passed by reference (such as `int *`) and that passing a Fortran character string will add the string's size as an invisible, final argument to a routine. The limited number of name-mangling schemes in particular makes it possible for tools such as GNU's `automake/autoconf` to automate mixed-language programming rules.

Interoperability with Fortran 90 is much more prob-

lematic because of the language's complexity (compared to FORTRAN 77) and as a result of the significant leverage given to compiler vendors in implementing the Fortran 90 standard. The areas in which interoperability is trickiest include varying name-mangling schemes across compilers for global data and procedures contained within a module, varying formats for the internal representation of pointer and allocatable arrays, and varying internal representation of derived types. Fortran vendors have yet to fully publish their implementations and define de facto standards in these three areas. As we'll see, the Fortran 2003 standard alleviates some of these obstacles but still falls short of providing full interoperability with C.

It's beyond this article's scope to address all the interoperability issues between Fortran 90/95 and C. Because our emphasis here is how to access derived types from C/C++, we assume henceforth that the exposed routines are either accessible from C using the `bind(c, name=...)` procedure attribute (for compilers supporting this Fortran 2003 feature) or have the exposed routines implemented outside a module statement, in which case the Fortran compiler's name-mangling is assumed to involve a single, appended underscore. As a further restriction, we assume the routines don't take optional arguments, although it's possible to pass `%null` in place of a missing, optional argument with some compilers. For simplicity, all methods acting on the object will be subroutines (not functions) and will pass the array size explicitly (meaning no dummy assumed shape arrays) when they expect an array argument. With these restrictions in mind, our starting point will be Fortran code:

```
module foo_mod
  type foo_type
    integer :: i
    real*8 :: r
    ! other data members follow
    ...
  end type foo_type
end module foo_mod
subroutine foo_dostuff(obj, r, i)
  use foo_mod
  implicit none
  type(foo_type) :: obj
  real*8 :: r
  integer :: i
  ! body of subroutine follows
  ...
end subroutine foo_dostuff
```

Our goal will be to expose routine `foo_dostuff`, whose first argument is object `obj` of type `foo_type`, to C.

Type Shadowing

Because derived types are functionally equivalent to C `structs`, the question naturally arises of whether we can directly access a derived type from C, as in

```
! Fortran
obj % i = 1234
obj % r = 9876.0

/* C */
typedef struct {
  int i;
  double r;
  /* other data members follow */
  ...
} foo_type;

foo_type obj;
obj.i = 1234;
obj.r = 9876.0;
```

In this case, each derived type member maps in memory to a corresponding member in the C `struct`. This approach is sometimes called *type shadowing* (<http://acts.neresc.gov/events/Workshop2003/slides/Rasmussen.pdf>).

For type shadowing to work, all members must be interoperable. Primitive types (real, integer, and so forth) including strings, with the caveat that strings in C are null (`'\0'`) terminated, are interoperable. `Logicals` tend to map to `ints` in C, but the value of truth varies across compilers (`-1` on some, `+1` on others). Nested or hierarchical derived types must recursively be interoperable, so a derived type that contains another derived type can be interoperable with C only if the contained derived types are also interoperable. A Fortran 90 derived type won't be interoperable if it contains an allocatable array or a pointer array (there's no equivalent of a Fortran pointer/allocatable array in C).

We define *deep* derived types as those that contain pointer or allocatable components; types without such components are *shallow*. We borrow this shallow/deep terminology from the Earth System Modeling Framework project (www.esmf.ucar.edu/download/index.shtml), which defines deep objects as those for which a constructor/destructor exists. A shallow object needn't be constructed or destroyed because its memory footprint is known at compile time—it's

Table 1. Support of Fortran 2003 interoperability features among some popular compilers.

Compiler/platform	bind(c)	iso_c_binding
gfortran 4.1.2 i686	No	No
pathf95 3.0 i686	No	No
g95 0.91 i686	Yes	Yes
pgf95 7.1-4 i686	Yes	Yes
ifort 10.1 i686	Yes	Yes
gfortran 4.1.1 x86_64	No	No
lf95 L8.00a x86_64	No	No
NAG f95 5.1 x86_64	Yes	Yes
pathf95 3.1 x86_64	No	No
ifort 9.1.041 ia64	Yes	No
xlf95 r 10.1.0.5 AIX64	Yes	Yes

shallow in the sense that its memory requirement is self-contained. A shallow type can contain any level of nested shallow types.

Note that the requirement for a derived type to be shallow is by no means sufficient to ensure its interoperability. Two more obstacles stand in the way: member reordering and padding. In C, member layout in memory is dictated by the order in which members are listed in the `struct` definition. Fortran compilers, on the other hand, are free to reorder members for performance benefits: a careful reordering can reduce the derived type's memory footprint while ensuring that members align to the bus size (typically 4 or 8 bytes). To minimize the need for padding, we recommend ordering members from largest to smallest—for instance, listing 8-byte reals followed by 4-byte integers and so forth. In some cases, however, padding can't be avoided. To prevent the need for the Fortran or C compiler to internally manipulate the derived type (or `struct`), we further recommend manual padding of the derived type's members. The following shallow derived types illustrate this technique (the first isn't interoperable, whereas the second is, even though it's functionally equivalent to the first):

```
! A non-interoperable derived type
type foo_type
  integer :: i4 ! 4 bytes
  real*8 :: r8 ! 8 bytes
  character(len=10) :: s10 ! 10 bytes
end type foo_type

! A derived type that is interoperable with
! typedef struct {
!   double r8;
!   int i4; int pad;
!   char s10[10];
! }foo_type;
type foo_type
```

```
sequence ! prevents member reordering
real*8 :: r8
integer :: i4
integer :: pad ! padding
character(len=10) :: s10
end type foo_type
```

Note that we can largely ignore the need for data member padding and reordering in Fortran 2003 thanks to the `bind(c)` keyword,⁴ which we can attach to the derived type declaration.⁵ In Fortran 2003, we would rewrite the previous code snippet as

```
! Fortran 2003 type that is interoperable with
! the C struct:
! typedef struct {
!   int i4;
!   double r8;
!   char s10[10];
! } foo_type;
use iso_c_binding
type, bind(c) :: foo_type
  integer(c_int) :: i4
  real(c_double) :: r8
  character(c_char) :: s10(10)
end type foo_type
```

The `sequence` statement is no longer required in this case. Note the statement `use iso_c_binding`, which imports the `kind` definitions of interoperable primitive types (`integer(c_int)`, `real(c_float)`, and the like). Table 1 provides a list of compilers that support Fortran 2003's `bind(c)` and `iso_c_binding` features. Keep in mind that compilers that don't yet support these features might do so in the near future.

We must emphasize here that `bind(c)` can't be applied to derived types containing allocatable/pointer array components. For the specific problem of exposing a derived type to C, Fortran 2003's new features don't provide a full solution—rather, they formalize what was known to work with most Fortran 90 compilers.

Opaque Containers

For deep derived types, a direct memory map can only be achieved provided we know the pointer/allocatable array descriptor's memory layout. The array descriptor, also known as the dope vector, typically contains information about the array's rank (number of dimensions), its starting

Table 2. Size in bytes of derived types with a pointer array member of rank 1, 2, and 3.*

Compiler/platform	array(:)	array(:, :)	array(:, :, :)	pointer container
gfortran 4.1.2 i686	24	36	48	4
pathf95 3.0 i686	56	68	80	32
g95 0.91 i686	40	52	64	4
pgf95 7.1-4 i686	96	120	144	4
ifort 10.1 i686	48	60	72	4
gfortran 4.1.1 x86_64	72	96	120	8
lf95 L8.00a x86_64	88	120	152	16
NAG f95 5.1 x86_64	64	88	112	8
pathf95 3.1 x86_64	96	120	144	48
ifort 9.1.041 ia64	96	120	144	8
xlF95_r 10.1.0.5 AIX64	80	104	128	16

*The array descriptor size typically increases by 12 bytes for each additional array dimension on 32-bit architectures (except pgf95) and by 24 bytes on 64-bit architectures (except lf95). For most compilers, the size of a derived type containing a single pointer member is that of an address (except for pathf95, lf95, and xlF95_r).

address, number of elements, and the stride in memory (a pointer array can be noncontiguous). The CHASM project aims to build a library of array descriptor formats implemented by compiler vendors.⁶ Maintaining such a library is a continuous effort, though, with descriptor formats varying between vendors and sometimes from one compiler version to another. Because CHASM no longer appears to be actively maintained, shadowing an allocatable/pointer array will likely remain a challenge for the foreseeable future.

One approach to overcome this problem involves accessing the Fortran type from C as an opaque object. Programmers frequently used this technique in the past to extend FORTRAN 77's limited memory management capability using C or in the context of the message-passing interface library (www.mpi-forum.org) to provide access to a C communicator object from FORTRAN 77; here, we apply it the other way around by extending C/C++ with Fortran. By opaque, we mean that the C code can't inspect the object's content—that is, it can't access the derived type members. However, the C code can hold onto the object and pass it to Fortran procedures, which can then freely access its content and modify its members. The restriction here is that all operations on the object must be via methods.

Consider, for instance, this deep derived type:

```
! A Fortran deep container
type foo_type
  real*8 :: sar(2)
  real, pointer :: arr(:)
end type_foo
```

We can wrap this derived type into an opaque object whose size is equal to or larger than the original derived type, thus we must estimate each member's size—for static arrays, it's the size of the type (8 bytes for `real*8`)

times the number of elements (two). For allocatable and pointer arrays, we must account for the descriptor size, which, at a minimum, will contain a starting address, a stride in memory, and the number of elements for each dimension or array rank. Array descriptors also typically contain rank-independent information.

Table 2 shows the descriptor size for arrays of rank 1, 2, and 3 and for various compilers. Although some exceptions exist, array descriptor size typically increases by 12 or 24 bytes for each additional dimension (rank); rank-independent storage ranges from 12 to 72 bytes, depending on vendor and architecture (32 versus 64 bits), so we expect our derived type code snippet to occupy up to 112 bytes. Allowing for some safety margin, we can then map the derived type to an opaque object of size 128 bytes (in this case, an array of characters). The opaque object's type doesn't matter—it can be anything as long as it's large enough to accommodate the derived type. We can then invoke method `foo_dostuff` from C by using code such as

```
/* Function prototype */
void foo_dostuff_(char *, double *, int *);
/* A C opaque object mapping to foo_type */
char foo_obj[128];
int i;
double r;
i = 1234;
r = 9876.0;
foo_dostuff_(foo_obj, &r, &i);
```

Using a Pointer to an Opaque Container

The need to determine a derived type's size can be cumbersome and error-prone: it varies according to the number of members, the padding algorithm the Fortran compiler uses, and our rough estimate of array descriptor size. We

can largely mitigate this problem by wrapping `foo_dostuff` with a routine that expects a pointer to the derived type (`foo_pointer_type`) rather than `foo_type`:

```
subroutine foo_dostuff_wrap(h, r, i)
  use foo_mod ! definition of foo_type
  implicit none
  type :: foo_pointer_type
    type(foo_type) :: ptr
  end type
  type(foo_pointer_type) :: h
  real*8 :: r
  integer :: i
  type(foo_type), pointer :: obj
  obj => h % ptr
  ! now call the method
  call foo_dostuff(obj, r, i)
end subroutine
```

The size of `foo_pointer_type` no longer depends on the number of members and their respective types, as in the previous example. For most compilers, the size of `foo_pointer_type` will be the size of an address (4 or 8 bytes), whereas other compilers might store additional information. This size appears in the fourth column of Table 2, with the greediest compiler requiring 48 bytes, or 12 elements of an integer array. Let's take this as our upper limit—we can now access subroutine `foo_dostuff_wrap` from C by using

```
void doo_dostuff(int*, double*, int*);
```

A variant of this approach involves modifying the Fortran interface so that the routine no longer expects a derived type but instead an array of integers. This array is then converted to `object_foo_pointer_type` using the intrinsic `transfer` function, a technique also used in the context of callback functions (http://macresearch.org/advanced_fortran_90_callbacks_with_the_transfer_function):

```
subroutine foo_dostuff_wrap2(ih, r, i)
  use foo_mod ! definition of foo_type
  implicit none
  type :: foo_pointer_type
    type(foo_type) :: ptr
  end type
  integer :: ih(12)
  real*8 :: r
  integer :: i
```

```
type(foo_pointer_type) :: h
type(foo_type), pointer :: obj
h = transfer(ih, h)
obj => h % ptr
! now call the method
call foo_dostuff(obj, r, i)
end subroutine
```

This version now expects only primitive type arguments, so we can invoke it straightforwardly from C and even from FORTRAN 77, which doesn't support derived types. Casting a derived type into a primitive type is particularly suitable for tools that parse Fortran source code (such as `f2py`) to generate cross-language bindings.

A Complete Example

Expanding on our previous examples, we're now ready to expose the code:

```
module foo_mod
  implicit none
  type :: foo_type
    character(len=10) :: string
    real*8, pointer :: array(:) =>null()
  end type foo_type
contains
  subroutine new(obj, str, n)
    ! Constructor
    type(foo_type) :: obj
    character(len=*), intent(in) :: str
    integer, intent(in) :: n
    obj % string = str
    allocate(obj % array(n))
  end subroutine new

  subroutine del(obj)
    ! Destructor
    type(foo_type) :: obj
    deallocate(obj % array)
  end subroutine del

  subroutine set_array(obj, arr)
    ! Array setter
    type(foo_type) :: obj
    real*8, intent(in) :: arr(:)
    obj % array = arr
  end subroutine set_array
end module foo_mod
```


to C using the pointer-to-opaque-handle technique from the previous section, in which we transfer the pointer to an integer array. The API consists of a deep derived type, a constructor, a destructor, and a setter routine. We define the pointer to the opaque type in a separate module for convenience:

```
module foo_pointer_mod
  use foo_mod
  type :: foo_pointer_type
    type(foo_type), pointer :: ptr
  end type foo_pointer_type
end module foo_pointer_mod
```

To overcome the module statement's name-mangling, we wrap each method with a subroutine, adding `foo` as a prefix to prevent name clashes:

```
subroutine foo_new(ih, str, n)
  use foo_mod, only : foo_type, new
  use foo_pointer_mod
  implicit none
  integer, intent(out) :: ih(12)
  character(len=*), intent(in) :: str
  integer, intent(in) :: n
  ! local variables
  type(foo_pointer_type) :: h
  ! create object
  allocate(h % ptr)
  ! now call constructor
  call new(h % ptr, str, n)
  ih = transfer(h, ih)
end subroutine foo_new

subroutine foo_del(ih)
  use foo_mod, only : foo_type, del
  use foo_pointer_mod
  implicit none
  integer, intent(inout) :: ih(12)
  ! local variables
  type(foo_pointer_type) :: h
  h = transfer(ih, h)
  ! now call destructor
  call del(h % ptr)
  deallocate(h % ptr)
  ih = 0
end subroutine foo_del
```

```
subroutine foo_set_array(ih, n, arr)
  use foo_mod, only : foo_type, set_array
  use foo_pointer_mod
  implicit none
  integer, intent(in) :: ih(12)
  integer, intent(in) :: n ! size of arr
  real*8, intent(in) :: arr(n)
  ! local variables
  type(foo_pointer_type) :: h
  h = transfer(ih, h)
  ! now call set_array
  call set_array(h % ptr, arr)
end subroutine foo_set_array
```

Note that `arr` is an explicit shape array in subroutine `foo_set_array`: we're passing its size as part of the list of arguments. Also, before using the pointer member `ptr` in type `foo_pointer_type`, we must first allocate it in `foo_new`. When no longer used, we then deallocate this pointer in `foo_del`.

We can get Python bindings for this code by using the `f2py` tool, which is capable of parsing Fortran code given that our routines only pass primitive types. `f2py` also compiles the code and produces the shared object, which we can load at runtime within Python. A typical Python session might look like this:

```
# Calling Foo from Python
from Foo import foo_new, foo_set_array,
foo_del
n = 10
# create and return handle
ih = foo_new('from Python', n)
# foo_set_array accepts lists
arr = [float(i) for i in range(n)]
foo_set_array(ih, arr)
# or numeric arrays
import numpy
foo_set_array(ih, numpy.array(arr))
# destroy object
foo_del(ih)
```

Other tools (such as SWIG) require access to the C code to define the interface. Let `foo.h` contain the corresponding C/C++ method signatures:

```
/* autotools settings */
#include <config.h>
```

```

/* name mangling */
#define FooNew FC_FUNC(foo_new, FOO_NEW)
#define FooDel FC_FUNC(foo_del, FOO_DEL)
#define FooSetArray\
FC_FUNC(foo_set_array, FOO_SET_ARRAY)


#ifdef __cplusplus
extern "C" {
#endif
void FooNew(int *ih, const char *str, const
int *n, size_t str_size);
void FooDel(int *ih);
void FooSetArray(const int *ih, const int *n,
const double *arr);
#ifdef __cplusplus
}
#endif

```

where `FC_FUNC` is a facility provided by `autoconf` and stored in `config.h` to handle the Fortran-to-C name-mangling. C++ is interoperable with C, provided the compiler is told that the C binding rule should be followed (`extern "C"`); our header file will work from both C and C++. When calling a Fortran routine with a `character(len=...)` dummy argument, the string's length must be the last argument (`size_t str_size` in `FooNew`). From `foo.h` alone, SWIG can generate the bindings for a host of languages, including Java, C#, OCaml, Perl, Ruby, Tcl, and Python.

Fortran stands out as the only programming language that doesn't (yet) have a formal and complete interoperable interface to C, with the latest 2003 standard only partially addressing this issue. Fortran developers seeking to expose their API to other languages must thus look for interoperability solutions outside the language's formal setting.

Whereas mixing FORTRAN 77 and C is straightforward, code produced with one Fortran 90 compiler doesn't readily interoperate with code produced by another. This puts a large burden on our build systems, which are required to compile libraries for every Fortran compiler flavor. We hope that the Fortran market will one day consolidate to the point where all compilers share the same module name-mangling scheme, use the same internal representation of a derived type, and offer the same representation for pointers and array descriptors. With compilers moving toward a common application binary interface, the

prospect of being able to mix different Fortran compilers in the same application could become reality. Until then, the different approaches presented here to expose Fortran derived types can help bridge the Fortran 90-to-Fortran 90 interoperability gap—that is, allow the mixing of multiple Fortran 90 compilers within a single application. 

References

1. T. Dahlgren et al., *Babel User's Guide*, Lawrence Livermore Nat'l Lab, 2004.
2. S. Kohn et al., "Divorcing Language Dependencies from a Scientific Software Library," *Proc. 10th SIAM Conf. Parallel Processing*, SIAM Press, 2001; <https://computation.llnl.gov/casc/components/docs/2001-siam-pp.pdf>.
3. C. Hill et al., "The Architecture of the Earth System Modeling Framework," *Computing in Science & Eng.*, vol. 6, no. 1, 2004.
4. M. Metcalf, J. Reid, and M. Cohen, *Fortran 95/2003 Explained*, Oxford Univ. Press, 2004.
5. J. Reid, "The Future of Fortran," *Computing in Science & Eng.*, vol. 5, no. 4, 2003, pp. 59–67.
6. C.E. Rasmussen et al., "CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability," *Proc. Los Alamos Computer Science Symp.*, 2001; <http://citeseer.ist.psu.edu/article/rasmussen01chasm.html>.

Alexander Pletzer is a research scientist at Tech-X. His research interests include solving partial differential equations and multi-language programming. Pletzer has a PhD in theoretical and plasma physics from the Australian National University. Contact him at pletzer@txcorp.com.

Douglas McCune is a computational scientist at the Princeton Plasma Physics Laboratory. His research interests include simulation of magnetically confined fusion plasmas and practical approaches to software componentization. McCune has an MS in computer science from Drexel University. Contact him at dmccune@pppl.gov.

Stefan Muszala is a computer scientist at Tech-X. His research interests include computer architecture and parallel and distributed computing. Muszala has a PhD in electrical and computer engineering from the University of Colorado, Boulder. Contact him at muszala@txcorp.com.

Srinath Vadlamani is a research scientist at Tech-X. His research interests include computational plasma physics, and dynamical systems. Vadlamani has a PhD in applied mathematics from the University of Colorado at Boulder. Contact him at srinath@txcorp.com.

Scott Kruger is a principal scientist at Tech-X. His research interests include fluid modeling of fusion plasmas and improved modeling via code coupling. Kruger has a PhD in nuclear engineering and engineering physics from the University of Wisconsin-Madison. Contact him at kruger@txcorp.com.