# LA-UR-24-25462

**Approved for public release; distribution is unlimited.**

| | |
|---|---|
| **Title:** | To Interoperability And Beyond: Interoperable Types Through the Promises of C and C++ and ABI Abuse |
| **Author(s):** | Solomon, Clell Jeffrey Jr. |
| **Intended for:** | Non-conference technical exchange. |
| **Issued:** | 2024-06-03 |



Los Alamos
NATIONAL LABORATORY

# To Interoperability And Beyond

## Interoperable Types Through the Promises of C and C++ and ABI Abuse

**Clell J. (CJ) Solomon**

13 June. 2024

# Abstract

This presentation presents a technique that allows passing of Fortran nested-type hierarchies interoperably to C++. The technique is motivated by the Eulerian Application Project's need to port code from Fortran to C++ to utilize the Kokkos performance portability library. The resulting method allows hierarchies of types to become interoperable while at the same time transforming Fortran array members of the original Fortran type into `Kokkos::View`s in the resulting C++ type.

# About Me and the Project

My Ph.D. was in Nuclear Engineering with a focus on variance reduction for computational Monte Carlo particle transport methods. I am currently the project leader of LANL's Eulerian Applications Project (EAP) where my technical focus is primarily on an unsplit hydrodynamics scheme that we are working on GPU offloading. I have a strong interest in software development and engineering, particularly in C++ where I enjoy leveraging template meta programming for performance gains and easing implementation hurdles.

The EAP focuses on providing a cell-based adaptive mesh refinement (CAMR) Eulerian-hydrodynamics simulation capability for high energy density (HED) physics simulations. The EAP code base consists of a 30+ year old Fortran backbone, but has been modernized relying on features up to Fortran 2018. We are currently working toward GPU acceleration of our algorithms via the Kokkos library.

# Disclaimers

- This presentation contains code of Fortran, C++, and Python flavors
- The code presented is “slide-ware”–details (hopefully not important ones) are omitted and best practices not always employed for brevity
- Some of the C++ examples (and even a Fortran one) uses templates
- If part of the code is unclear to you, ask me to explain

# Outline



# Outline



# Interoperability of Modern Fortran and C++ is Challenging

- The EAP code base makes heavy use of Fortran derived types
  - Organize related data
  - Pass data explicitly through API calls with managed `intent` rather than global `use` statements
- Often derived types are nested within other derived types
- Many of these types have become “vocabulary types” of the code
  - Types that are utilized throughout the code base and all developers should know about them
  - Common examples are the AMR mesh type `mesh_t` and its subcommponents:
    - `cells_t`—holds cell counts, global indices, neighbor information, etc.
    - `levels_t`—holds information about AMR levels, most importantly what is as the “top-level”, i.e., active
- This presentation addresses how the EAP made such types more accessible in C++

# Straw Man: Want to Use the Following in C++

```
type :: cells_t
  integer, dimension(:), allocatable :: cell_address
end type

type :: levels_t
  integer :: numtop
  integer, dimension(:), allocatable :: ltop
end type

type :: mesh_t
  type(cells_t) :: cells
  type(levels_t) :: levels
end type
```

# Straw Man: Use Case As Follows

```
interface
  subroutine some_physics(mesh) bind(c)
    type(interop_mesh_t) :: mesh
  end subroutine
end interface

type(mesh_t) :: mesh

call some_physics(make_interoperable(mesh))
```

```
extern "C" {

void some_physics(
  in_t<interop_mesh_t> const& iomesh ) {
  auto mesh = map(iomesh, transform_fn{})

  Kokkos::parallel_for(/*...*/, [=](int i){
    /* use mesh.cells and mesh.levels */
  });
}

} // extern "C"
```

# Goal: Provide Equivalent Structured Types in C++

- Desire to “mirror” these vocabulary types to our C++
  - Would be trivial if all data in the types were interoperable per `iso_c_binding`
  - Of course they aren’t—consider

    ```
    integer, dimension(:), allocatable :: cell_address
    ```
- Once on the C++ side we want the underlying representation of arrays to be easily changeable
  - We are using `Kokkos::Views` for now—easy interchange from host side versus device side views
  - We should not preclude (design out of) other possibilities in the future
- Allow for selective interoperability
  - Not all data in a Fortran derived type needs to be mirrored
- If the data should be immutable, then make sure it is `const`-correct

# The Real Problems to Be Addressed

1. How do we **represent** array data interoperably?
2. How do we give staff the ability to quickly generate interoperable types that
   - Are easy to change
   - Maintain `const`-correctness/`intent(in)`-ness
   - Allow simple transformations of the structures while maintaining structured order

The remainder of this talk will address how the EAP codes are solving these problems.

# Outline



# What is C/Fortran Interoperabilty?

- The ability to pass data objects from C to Fortran and/or Fortran to C and maintain the way in which they are **interpreted**
- Essentially this requires that types, both basic and derived, have the same binary layout **including padding and alignment requirements**
- Fortran **does not** require that the order of elements in a derived type be the definition order–free to optimize by rearrangement
- C (and C++ by virtue of C compatiblity) **does** require that the order of elements in a derived type be the definition order

```c
struct s {
  int i;
  double d;
  char c;
};

constexpr auto size_s =
  sizeof(s); /* 24 bytes */
constexpr auto align_s =
  alignof(s); /* 8 bytes */
```

# This is ABI Compatibility!

- As software engineers we commonly deal with Application Programming Interfaces (APIs)
  - How we express interfaces to isolated components of code
  - Function calls, class methods, overload sets, etc.
- The Application Binary Interface (ABI) is usually a job left to the compiler
  - When a variable is passed to a function, how much stack space must be allotted?
  - Is the variable passed by value or reference?
- Type systems are the programmer-facing ABI requirement
  - If a type is passed, be it fundamental or derived, then if the same type is received there **should not** be an ABI incompatibility
  - Compilers **can and do** enforce this—cannot pass a `real(real32)` to a `real(real64)` in Fortran

When we jump a language barrier there is only so much a compiler can do to help us—they cannot type check in this instance!

# With Fortran 2018 Arrays Are Interoperable...

- ...when being passed to functions as deferred shape arrays

```
interface
  subroutine my_c_function(a) bind(c)
    real(c_double), dimension(:,:) :: a ! OK :-D
  end subroutine
end interface
```

- **NOT** when members of a type

```
type, bind(c) :: my_type
  real(c_double), dimension(:,:) :: a ! NOT OK :-(
end type
```

- On the C++-side we can receive the array as a `CFI_cdesc_t`

```
#include <ISO_Fortran_binding.h>
extern "C" {
  void my_c_funtion(CFI_cdesc_t const& a) // <- Fortran is pass-by-reference, ABI :-O
  { /* Do stuff with a*/ }
}
```

# What is this `CFI_cdesc_t`?

- An **implementation dependent** representation of a Fortran array
- A fancy `void*`
- A dope vector:

  *In computer programming, a dope vector is a data structure used to hold information about a data object, especially its memory layout.*

  – `https://en.wikipedia.org/wiki/Dope_vector`

# The GNU `CFI_cdesc_t` Implementation

```c
/* CFI_dim_t. */
typedef struct CFI_dim_t
  {
    CFI_index_t lower_bound;
    CFI_index_t extent;
    CFI_index_t sm;
  }
CFI_dim_t;

/* CFI_cdesc_t, C descriptors are cast to this structure
   as follows:
   CFI_CDESC_T(CFI_MAX_RANK) foo;
   CFI_cdesc_t * bar = (CFI_cdesc_t *) &foo;
 */
typedef struct CFI_cdesc_t
 {
    void *base_addr;
    size_t elem_len;
    int version;
    CFI_rank_t rank;
    CFI_attribute_t attribute;
    CFI_type_t type;
    CFI_dim_t dim[];
 }
CFI_cdesc_t;
```

```c
/* CFI_CDESC_T with an explicit type. */
#define CFI_CDESC_TYPE_T(r, base_type) \
        struct { \
                base_type *base_addr; \
                size_t elem_len; \
                int version; \
                CFI_rank_t rank; \
                CFI_attribute_t attribute; \
                CFI_type_t type; \
                CFI_dim_t dim[r]; \
        }
#define CFI_CDESC_T(r) CFI_CDESC_TYPE_T (r, void)
```

# Outline



# Want An Interoperable Array Representation that Can be Stored in a Type

- Solution: Create our own version of a dope vector
- Flattens the implementation dependent bits to the common set of things we want/need
- Removes superfluous (at least to us) information
- Maintains interoperability—what we're after in the first place
- Drops knowledge of the data type being referenced but maintains rank knowledge
- Unfortunately, due to Fortran's lack of templating capability this requires a fair amount of code to be be consistently stamped out—we auto generate it with a Python script

# EAP's dope Types

- We generate a dope type for each rank, e.g.,:

```
type, bind(c) :: dope0
  type(c_ptr) :: base_addr
  integer(c_size_t) :: elem_len
  integer(c_int64_t) :: rank
  integer(c_int64_t) :: type
end type

type, bind(c) :: dope1
  type(c_ptr) :: base_addr
  integer(c_size_t) :: elem_len
  integer(c_int64_t) :: rank
  integer(c_int64_t) :: type
  type(dim_t), dimension(1) :: dim
end type
```

```
type, bind(c) :: dope2
  type(c_ptr) :: base_addr
  integer(c_size_t) :: elem_len
  integer(c_int64_t) :: rank
  integer(c_int64_t) :: type
  type(dim_t), dimension(2) :: dim
end type

type, bind(c) :: dope3
  type(c_ptr) :: base_addr
  integer(c_size_t) :: elem_len
  integer(c_int64_t) :: rank
  integer(c_int64_t) :: type
  type(dim_t), dimension(3) :: dim
end type
```

# Quick Aside: If Only There Were More Compiler Uniformity

- Fortran actually does have a minimal amount of type-templating capability—parametrized types

```
type, bind(c) :: dope(R)
  integer, dim :: R
  type(c_ptr) :: base_addr
  type(size_t) :: elem_len
  type(c_int64_t) :: rank
  type(c_int64_t) :: type
  type(dim_t), dimension(R) :: dims
end type
```

- Intel and GCC Fortran compilers allowed the above code, but IBM's compiler doesn't allow interoperable types to be parametrized
- If this would work it would drastically cut down on the amount of code that has to be generated

# EAP's `make_dope` Function

- A single overloaded `make_dope` function returns the appropriate `dope` type for arrays
- `make_dope` only copies meta-data for the array, not the array itself
- Restriction: arrays must contain interoperable data types

```
interface
  subroutine c_physics(int_data, real_data) &
    bind(c)
    type(dope1) :: int_data
    type(dope2) :: real_data
  end subroutine
end interface
```

```
integer :: i
integer, dimenion(10) :: int_data
real(real64), dimenion(10,10) :: real_data

call c_physics(make_dope(int_data), &
               make_dope(real_data))
```

Importantly, the `dope` types can be stored in other interoperable types!!!

# Templated `dope` Types in C++

- In C++ we can now have a templated `dope<T,R>`
- Because C++ guarantees the same ordering requirement as C and that inherited data comes first then, if we are careful, we have the same required ABI layout
- Adding the type and rank information does not add to the actual size, but adds the ability to use this information in the C++

```
1  template<typename T>
2  struct dope_base {
3    T* base_addr;
4    size_t elem_len;
5    ptrdiff_t r;
6    ptrdiff_t t;
7  };
8  template<typename T, size_t R>
9  struct dope : detail::dope_base<T> {
10   struct dim_t {
11     ptrdiff_t lower_bound;
12     ptrdiff_t extent;
13     ptrdiff_t sm;
14   };
15   dim_t dim[R];
16 };
17 template<typename T>
18 struct dope<T,0> : detail::dope_base<T> {};
```

# Turning `dope<T,R>`s into `Kokkos::View`s

- Because the type and rank information is encoded in the `dope<T,R>`, it is possible to write a generic function to convert any `dope<T,R>` to a `Kokkos::View`
- Putting it all to together, from Fortran to Kokkos looks like the following:

Fortran

```fortran
interface
  subroutine c_physics(int_data, real_data) &
    bind(c)
    type(dope1) :: int_data
    type(dope2) :: real_data
  end subroutine
end interface

call c_physics(make_dope(int_data), &
               make_dope(real_data))
```

C++

```cpp
extern "C" {
void c_physics(dope<int,1> const& int_data
               dope<double,2> const& real_data) {
  auto int_view = to_kokkos_view(int_data);
  auto real_view = to_kokkos_view(real_data);

  Kokkos::parallel_for(/* ... */, [=](int i){
    /* do things with views */
  });
}
} // extern "C"
```

# Outline



# Our Previous Process For Building Interoperable Types was Error Prone and Difficult to Change

- Consider

```
type :: levels_t
  integer :: numtop
  integer, dimension(:), allocatable :: ltop
end type
```

- First, manually construct the interoperable type

```
type :: interop_levels_t
  integer :: numtop
  type(dope1) :: ltop
end type
```

```
struct interop_levels_t {
  int numtop;
  dope<int,1> ltop;
};
```

- Then, manually write a parallel class using `Kokkos::Views`

```
class levels_t {
  int numtop;
  Kokkos::View<integer*, Kokkos::HostSpace> :: ltop

  levels_t(interop_levels_t const& iolevels) { /* details */ }
}
```

# Problems with The Current Approach

- Tedious
  - All the interoperable Fortran and C++ types must be written by hand
  - All the parallel C++ types must be written by hand
  - If I want a representation that now has the views on the device I have to write that one too
  - I need different versions depending on if the data is mutable or immutable
- Error Prone
  - If the interoperable Fortran and C++ types aren't kept in the same order, you have an ABI incompatibility that can be difficult to track
  - Original types could get out of sync with interoperable types
- Inconsistency across people doing porting work on how this is accomplished

# Automatic Generation of Interoperable Types

- EAP has written infrastructure where the interoperable types can be described with Python and the interoperable Fortran type and C++ are automatically generated

```
use gentypes

class cells_t:
    cell_address = gentypes.array(gentypes.integer, 1)
class levels_t:
    numtop = gentypes.by_reference(gentypes.integer)
    ltop = gentypes.array(gentypes.integer,1)
class mesh_t:
    cells = cells_t
    levels = levels_t
```

- Interoperable Fortran and C++ types guaranteed to be generated consistently (avoids ABI issues)
- All names enforced to match (compile error) original type and member names
- Bonus: Use of these interoperable descriptors in Python allows for selective and consistent inclusion/exclusion of members

# Problems with The Current Approach

- Tedious
  - ~~All the interoperable Fortran and C++ types must be written by hand~~ (kind of—the Python is a lot more compact)
  - All the parallel C++ types must be written by hand
  - If I want a representation that now has the views on the device I have to write that one too
  - I need different versions depending on if the data is mutable or immutable
- Error Prone
  - ~~If the interoperable Fortran and C++ types aren't kept in the same order, you have an ABI incompatibility that can be difficult to track~~
  - Original types could get out of sync with interoperable types
- ~~Inconsistency across people doing porting work on how this is accomplished~~

# How to Receive the Interoperable Type on the C++ Side?

- Clearly, one wants something like

```
1 struct interop_levels_t {
2   dope<int,0> numtop;
3   dope<int,1> ltop;
4 };
```

- Doing the above would make automatic conversion of `dope<T,R>`s to `Kokkos::Views` challenging
  - No compile-time way to iterate the members of a `struct` (yet...maybe in C++26)
- Could auto-generate it with more scripting, but C++ has a way to auto-generate code: templating
- Define our own `structure` type that provides for **compile-time iteration over its members**

# Why Emphasize the Compile-Time Iteration

- The types of things in C++ are merely compile time constructs the help enforce ABI requirements
- If one can iterate over the **member types** of a structure, then one can generate a structure with equivalent member names but **different member types** *WITHOUT* explicitly writing the type

```
1 struct interop_levels_t {
2   int numtop;
3   dope<int,1> ltop;
4 };
```

⇒

```
1 struct levels_t {
2   const int numtop;
3   Kokkos::View<const int*> ltop;
4 };
```

- One can also obtain and maintain `const`-correctness of data *WITHOUT* explicitly writing more types

# The EAP `structure` Type

- The details of implementing the `structure` are beyond the scope of this presentation
  - 200 lines using moderately advanced C++ template and template meta programming techniques)
- At the end of the day it provides for the following use case

```
EAP_STRUCTURE_MEMBER(numtop)
EAP_STRUCTURE_MEMBER(ltop)
using interop_levels_t = structure<numtop<int>,ltop<dope<int,1>>>;
```

which is equivalent in size, layout, and alignment (i.e., ABI compatible) with

```
struct interop_levels_t {
  dope<int,0> numtop;
  dope<int,1> :: ltop;
};
```

# An Entire `structure` Can Be Mapped Recursively

- `structure`s can be mapped to structures of the same description (same member names) but with different types
- Consider the following:

```
1  class transform_fn {
2    template<typename T>
3    T operator()(T const& v) const { return v; }
4
5    template<typename T, size_t R>
6    auto operator()(dope<T,R> const& v) const { return to_kokkos_view(v); }
7  };
8  extern "C" {
9  void c_physics(interop_levels_t const& iolevs) {
10   auto levs = map(iolevs, transform_fn{}); // <- the types are converted
11 }
12 } // extern "C"
```

- The `map` function will take `levs` and build a new `structure` type where all the `dope<T,R>`s are transformed to `Kokkos::View`s and everything else is left alone
- The `map` function works recursively if there are `structure`s within `structure`s

# `structure`s Can Be Immutable

- There is a challenge solving the immutability problem
  - In Fortran, an `intent(in)` array's data is immutable
  - In C++, a `Kokkos::View<double*> const`'s data is **mutable**–it has reference semantics
  - A `Kokkos::View<double const*>`'s data is **immutable**
- Similarly `dope<double const, 1>` indicates a dope vector to immutable data
- However, the auto-generation code for the types should not specify whether a type is mutable or immutable because it may need to be one or the other depending on context
- Instead, provide an `in_t<T>` that, given a `structure` or `dope<T,R>`, returns a type with **data** immutable versions
- Example:

```
1  void some_physics(in_t<interop_mesh_t> const& iomesh {
2    auto mesh = map(iomesh, transform_fn{}); // <- resulting Kokkos::Views have immutable data
3  }
```

# Problems with The Current Approach

- Tedious
  - ~~All the interoperable Fortran and C++ types must be written by hand~~ (kind of—the Python is a lot more compact)
  - ~~All the parallel C++ types must be written by hand~~
  - ~~If I want a representation that now has the views on the device I have to write that one too~~
  - ~~I need different versions depending on if the data is mutable or immutable~~
- Error Prone
  - ~~If the interoperable Fortran and C++ types aren't kept in the same order, you have an ABI incompatibility that can be difficult to track~~
  - ~~Original types could get out of sync with interoperable types~~—compile-time error
- ~~Inconsistency across people doing porting work on how this is accomplished~~

# Outline



# An EAP Developer Want To Use

```
1  type :: cells_t
2    integer, dimension(:), allocatable :: cell_address
3  end type
4
5  type :: levels_t
6    integer :: numtop
7    integer, dimension(:), allocatable :: ltop
8  end type
9
10 type :: mesh_t
11   type(cells_t) :: cells
12   type(levels_t) :: levels
13 end type
```

# They Write (Okay, Plus a Little CMake)

```
use gentypes

class cells_t:
    cell_address = gentypes.array(gentypes.integer, 1)
class levels_t:
    numtop = gentypes.by_reference(gentypes.integer)
    ltop = gentypes.array(gentypes.integer,1)
class mesh_t:
    cells = cells_t
    levels = levels_t
```

From this, interoperable types are generated for them as part of the build.

# They Use It

```f90
interface
  subroutine some_physics(mesh) bind(c)
    use interop_mesh_types
    type(interop_mesh_t) :: mesh
  end subroutine
end interface

type(mesh_t) :: mesh

call some_physics(make_interoperable(mesh))
```

```cpp
#include "interop_mesh_types.hh"

extern "C" {

void some_physics(
  in_t<interop_mesh_t> const& iomesh ) {
  auto mesh = map(iomesh, transform_fn{});

  Kokkos::parallel_for(/*...*/, [=](int i){
    /* use mesh.cells and mesh.levels */
  });
}

} // extern "C"
```

# Questions