

Interoperation of Fortran with C

Area of semantics	within Fortran	within C
function (procedure) call	invoke C function or interoperable Fortran procedure	invoke interoperable Fortran procedure
main program	only one: either Fortran or C	
intrinsic data types	subset of Fortran types denoted as interoperable; not all C types are known	not all Fortran types may be known
derived data types	special attribute enforces interoperability with C struct types	„regular“ Fortran derived types not (directly) usable
global variables	access data declared with external linkage in C	access data stored in COMMON or module variable
dummy arguments	arrays or scalars	pointer parameters
dummy arguments	with VALUE attribute	non-pointer parameters

■ Dealing with I/O:

- Fortran record delimiters
- STREAM I/O already dealt with

■ Earlier attempts

- F2C interface
- `fortran.h` include file
- proprietary directives

are not discussed in this course

Focus here is on:
standard conforming
Fortran/C interoperability

■ C and Fortran pointers

- different concepts!
- partial semantic overlap
- procedure/function pointers

Semantics	within Fortran	within C
C pointer	object of <code>TYPE(c_ptr)</code>	<code>void *</code>
C function pointer	object of <code>TYPE(c_funptr)</code>	<code>void (*)()</code>

- module functions are provided via an intrinsic module to map data stored inside these objects to Fortran `POINTERS` and procedure pointers

■ Used for implementing C interoperable types, objects and functions

- it must be possible to describe function interfaces via a C prototype

■ Companion may be

- a C processor
- another **Fortran processor** supporting C interoperation
- or **some other** language supporting C interoperation

■ Note:

- different C processors may have different ABIs and/or calling conventions
- therefore not all C processors available on a platform may be suitable for interoperation with a given Fortran processor

Example program:

```
PROGRAM myprog
  USE, INTRINSIC :: iso_c_binding
  INTEGER(c_int) :: ic
  REAL(c_float) :: rc4
  REAL(c_double), ALLOCATABLE :: a(:)
  CHARACTER(c_char) :: cc
  :
  ALLOCATE(a(ic), ...)

  CALL my_c_subr(ic,a)
  :
END PROGRAM
```

a module provided by
the Fortran processor

further stuff omitted here –
will be shown later

might be implemented
in Fortran or C.
Will show a C implementation later

■ via KIND parameters

- integer constants defined in ISO_C_BINDING intrinsic module

C type	Fortran declaration	C type	Fortran declaration
int	integer(c_int)	char	character(len=1 ,kind=c_char)
long int	integer(c_long)		
size_t	integer(c_size_t)		
[un]signed char	integer(c_signed_char)	_Bool	logical(c_bool)
float	real(c_float)		
double	real(c_double)		

Annotations:

- Callout for `c_long`: may be same as `c_int`
- Callout for `len=1`: may be same as `kind('a')`
- Callout for `real(c_float)`: on x86 architecture: the same as default real/double prec. type. But this is not guaranteed

- a **negative** value for a constant causes compilation failure (e.g., because no matching C type exists, or it is not supported)
- a standard-conforming processor must only support `c_int`
- compatible C types derived via typedef also interoperate

Calling C subprograms from Fortran: a simple interoperable interface

■ Assume a C prototype

```
void My_C_Subr(int, double []);
```

or double * ?

- C implementation not shown

■ Need a Fortran interface

- explicit interface

■ BIND(C, name='...') attribute

- suppress Fortran name mangling
- label allows mixed case name resolution and/or renaming (no label specified → lowercase Fortran name is used)
- cannot have two entities with the same binding label

left-out bits from previous program

```
INTERFACE
  SUBROUTINE my_c_subr(i, d) &
    BIND(C, NAME='My_C_Subr')
  USE, INTRINSIC :: iso_c_binding

  INTEGER(c_int), value :: i
  REAL(c_double) :: d(*)
END SUBROUTINE my_c_subr
END INTERFACE
```

■ VALUE attribute/statement

- create copy of argument
- some limitations apply (e.g., cannot be a POINTER)

■ Scalar vs. array pointers

- no unique interpretation in C
- check API documentation

■ C function with `void` result

- may interoperate with a Fortran **subroutine**

■ All other C functions

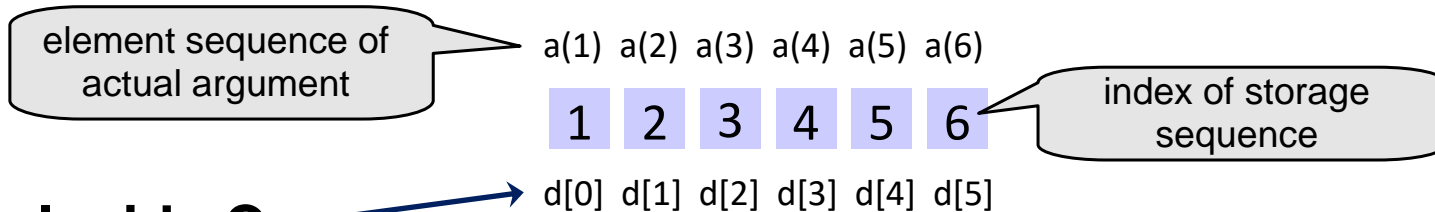
- may interoperate with a Fortran **function**

■ Link time considerations

- **recommendation:** perform linkage with Fortran compiler driver to assure Fortran RTL is linked in
- may need a special compiler link-time option if main program is in C (this is processor-dependent)

Return to previous example:

- assume that six array elements have been allocated
- remember layout in memory: **contiguous** storage sequence



Inside C

- formal parameter `double d[]` uses zero-based indexing (C ignores **any** lower bound specification in the Fortran interface!)

Note:

- in a call from Fortran, a non-contiguous array (e.g. a section) may be used → will be automatically compactified (copy-in/out)
- need to do this manually in calls from C

■ Example Fortran interface

```
INTERFACE
  SUBROUTINE solve_mat( &
    b, n1, n2) BIND(C)
  USE, INTRINSIC :: iso_c_binding
  INTEGER(c_int), VALUE :: n1, n2
  REAL(c_double) :: b(n1,*)
END SUBROUTINE solve_mat
END INTERFACE
```

■ Two possible C prototypes

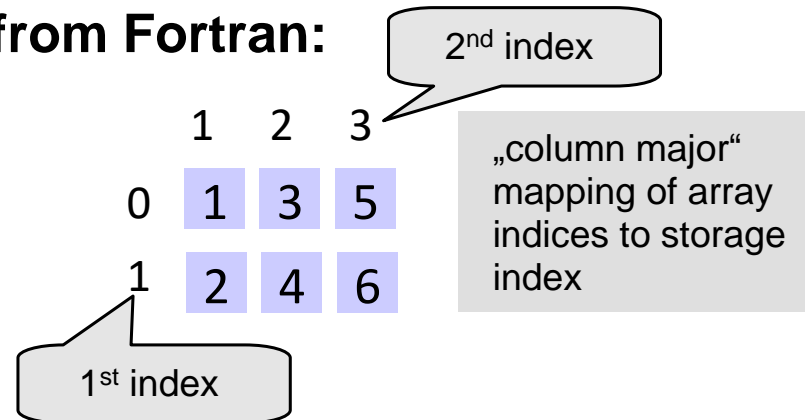
```
void solve_mat(double *, \
               int, int);
```

```
void solve_mat(double [][*], \
               int, int);
```

C99 VLA (variable length array)

■ Assume actual argument in call from Fortran:

```
DOUBLE PRECISION :: rm(0:1,3)
:
CALL solve_mat(rm, 2, 3)
```



■ First alternative – manual mapping

- example implementation:

```
void solve_mat(double *d, int n1, int n2) {  
    double **dmap;  
    int i, k;  
    dmap = (double **) malloc(n2 * sizeof(double *));  
    for (i=0; i<n2; i++) {  
        dmap[i] = d + n1 * i;  
    }  
    // now access array elements via dmap  
    for (k=0; k<n1; k++) {  
        dmap[i][k] = ...;  
    }  
    ...  
    free (dmap);  
}
```

force **dmap to contiguous storage layout

LHS is of type double

1st index

	0	1	2
0	1	3	5
1	2	4	6

„row major“
mapping of array
indices to storage
index

2nd index

■ Second alternative – C99 VLA

- example implementation:

```
void solve_mat(double d[][n1], int n1, int n2) {
    int i, k;
    // directly access array elements
    for (i=0; i<n2; i++) {
        for (k=0; k<n1; k++) {
            d[i][k] = ...;
        }
    }
    ...
}
```

LHS is of type double

1st index

„row major“
mapping of array
indices to storage
index

2nd index

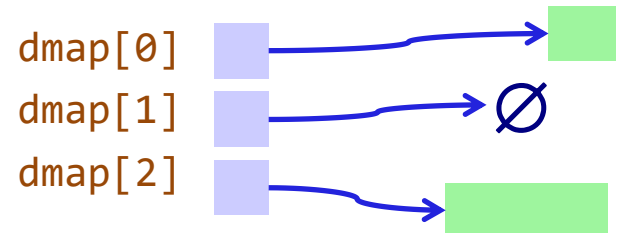
	0	1	2
0	1	3	5
1	2	4	6

■ Caveat for use of ** (pointer-to-pointer):

- **in general** this describes a non-contiguous storage sequence → **cannot** be used as interoperable array parameter

```
dmap[i] = (double *) malloc(...);
```

double **dmap;



Handling of strings (1)

- Remember: character length must be 1 for interoperability
- Example: C prototype

```
int atoi(const char *);
```

- matching Fortran interface declares `c_char` entity an assumed size **array**

```
INTERFACE
  INTEGER(c_int) FUNCTION atoi(in) bind(c)
    USE, INTRINSIC :: iso_c_binding
    CHARACTER(c_char), DIMENSION(*) :: in
  END FUNCTION
END INTERFACE
```

Handling of strings (2)

■ Invoked by

```
USE, INTRINSIC :: iso_c_binding
CHARACTER(LEN=: ,KIND=c_char), ALLOCATABLE :: digits

ALLOCATE(CHARACTER(LEN=5) :: digits)
digits = c_char_'1234' // c_null_char

i = atoi(digits)    ! i gets set to 1234
```

C string needs terminator

- **special exception** (makes use of storage association):
actual argument may be a scalar character string

■ Character constants in ISO_C_BINDING with C-specific meanings

Name	Value in C
c_null_char	'\0'
c_new_line	'\n'
c_carriage_return	'\r'

most relevant subset

C Interoperation with derived types

■ Example:

```
USE iso_c_binding
:
TYPE, BIND(C) :: dtype
  INTEGER(c_int) :: ic
  REAL(c_double) :: d(10)
END TYPE dtype
```

is interoperable with

```
typedef struct {
  int i;
  double dd[10];
} dtype_c;
```

and typed variables can be used
e.g., in argument lists

■ Notes:

- naming of types and components is irrelevant
- `bind(c)` cannot have a label in this context. It cannot be specified together with `sequence`
- position of components must be the same
- type components must be of interoperable type

- **In this context, Fortran type components must **not** be**
 - pointers or allocatable
 - zero-sized arrays
 - type bound procedures
- **Fortran type must not be**
 - extension of another type (and an interoperable type cannot itself be extended!)
- **C types which cannot interoperate:**
 - union types
 - structs with bit field components
 - structs with a flexible array member

Handling non-interoperable data – the question now is ...

- when and how to make objects of the (**non-interoperable!**) Fortran type

```
TYPE :: fdyn
  REAL(c_float), ALLOCATABLE :: f(:)
END TYPE fdyn
```

array size implicit

available within C

- when and how to make objects of the analogous C type

```
typedef struct cdyn {
  int len;
  float *f;
} Cdyn;
```

available within Fortran

Case 1: Data only accessed within C

API calls are

```
Cdyn *Cdyn_create(int len) {
    this = (Cdyn *) malloc(...);
    this->f = (float*) malloc(...);
    return this;
}
void Cdyn_add(Cdyn *v, ...) {
    :
    v->f[i] = ...;
    :
}
```

copy of v produced at invocation

Assumptions:

- want to call from Fortran
- but no access to type components needed within Fortran

Required Fortran interface

```
USE, INTRINSIC :: iso_c_binding
INTERFACE
    TYPE(c_ptr) FUNCTION &
        cdyn_create(len) bind(c,...)
    IMPORT :: c_int, c_ptr
    INTEGER(c_int), VALUE :: len
END FUNCTION
SUBROUTINE cdyn_add(h, ...) &
    bind(c,...)

    IMPORT :: c_ptr
    TYPE(c_ptr), value :: h
    :
END SUBROUTINE
END INTERFACE
```

object of type **c_ptr** requires value attribute here

■ Opaque derived types defined in ISO_C_BINDING:

- `c_ptr`: interoperates with a `void *` C object pointer
- `c_funptr`: interoperates with a C function pointer.

■ Useful named constants:

- `c_null_ptr`: C null pointer
- `c_null_funptr`: C null function pointer

```
TYPE(c_ptr) :: p = c_null_ptr
```

■ Logical module function that checks pointer association:

- `c_associated(c1[,c2])`
- value is `.false.` if `c1` is a C null pointer or if `c2` is present and points to a different address. Otherwise, `.true.` is returned
- typical usage:

```
TYPE(c_ptr) :: res

res = get_my_ptr( ... )
IF (c_associated(res)) THEN
  : ! do work with res
ELSE
  STOP 'NULL pointer produced by get_my_ptr'
END IF
```

```
USE, INTRINSIC :: iso_c_binding
:
TYPE(c_ptr) :: handle
:
handle = cdyn_create(5_c_int)
IF (c_associated(handle)) THEN
  CALL cdyn_add(handle,...)
END IF
CALL cdyn_destroy(handle)
```

all memory
management done in C

■ Typeless „handle“ object

- because objects of (nearly) any type can be referenced via a `void *`, no matching type declaration is needed in Fortran

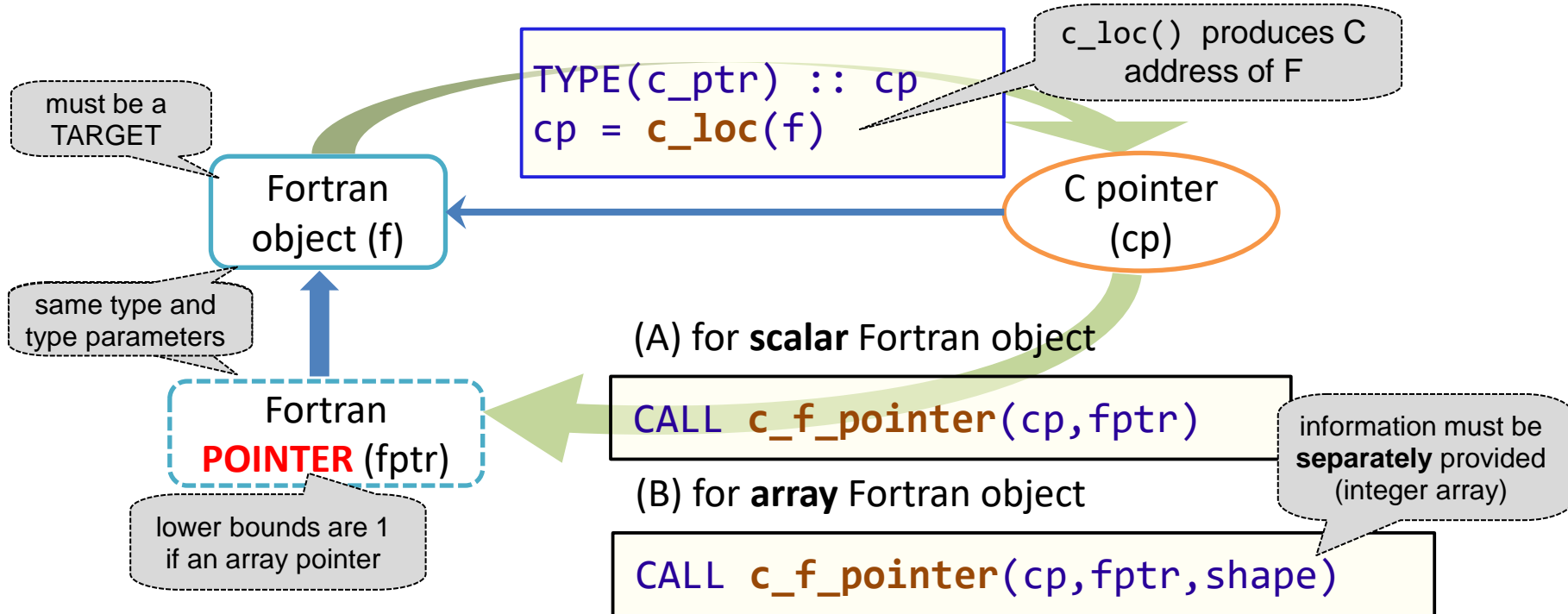
■ Design problem:

- no disambiguation between different C types is possible → loss of type safety

see exercise for a
possible solution

Setting up a mapping between a Fortran object and a C pointer

Module ISO_C_BINDING provides module procedures



- pointer association (blue arrow) is set up as a result of their invocation (green arrows)

1. Fortran object is of **interoperable** type and type parameters:

- variable with **TARGET** attribute,
- or allocated variable with **TARGET** attribute, non-zero length,
- or associated scalar pointer

in scenario 1, the object might also have been created within C (Fortran target then is anonymous). In any case, the data can be accessed from C.

2. Fortran object is a **non-interoperable**, non-polymorphic **scalar** without length type parameters:

- non-allocatable, non-pointer variable with **TARGET** attribute,
- or an allocated allocatable variable with **TARGET** attribute,
- or an associated pointer.

nothing can be done with such an object within C

Case 2: Data accessed only within Fortran

Fortran Library definition

```

MODULE mylib
  USE, INTRINSIC :: &
      iso_c_binding
  TYPE :: fdyn
    REAL, ALLOCATABLE :: f(:)
  END TYPE FDYN
CONTAINS
  : ! continued to the right

```

```

TYPE (c_ptr) FUNCTION &
    fdyn_create(len) BIND(C,...)
  INTEGER(c_int), VALUE :: len
  TYPE(fdyn), POINTER :: p
  ALLOCATE(p)
  ALLOCATE(p%f(len))
  fdyn_create = c_loc(p)
END FUNCTION
END MODULE mylib

```

scenario 2

- noninteroperable derived data type
- provide an interoperable constructor written in Fortran

Pointer goes out of scope

- but target remains reachable via function result

C prototype:

```
void *fnew_stuff(int);
```

Case 2 (cont'd): Retrieving the data

Client code in C:

```
void *fhandle;
int len = 5;

fhandle = Fdyn_create(len);

Fdyn_print(fhandle);
```

do not try to
dereference handle
except for NULL check

- can have multiple handles to different objects at the same time (thread-safeness)
- again no matching type needed on client
- require Fortran implementation of `Fdyn_print()`

... here it is:

```
SUBROUTINE fdyn_print(h) BIND(C,...)
  TYPE(c_ptr), VALUE :: h
  TYPE(fdyn), POINTER :: p

  CALL c_f_pointer(h, p)
  IF (allocated(p%f)) THEN
    WRITE(*, FMT=...) p%f
  END IF
END SUBROUTINE
```

scenario 2

- ... and must not forget to
 - implement „destructor“ (in Fortran)
 - and call it (from C or Fortran) for each created object
- to prevent memory leak**

■ With these functions,

- it is possible to subvert the type system (**don't** do this!)
(push in object of one type, and extract an object of different type)
- it is possible to subvert rank consistency (**don't** do this!)
(push in array of some rank, and generate a pointer of different rank)

■ Implications:

- implementation-dependent behaviour
- security risks in executable code

■ Recommendations:

- use with care (testing!)
- encapsulate use to well-localized code
- don't expose use to clients if avoidable

- We haven't gone the whole way towards **fully** solving the problem
 - won't actually do so in this talk
- Return to Case 1:

```
typedef struct Cdyn {  
    int len;  
    float *f;  
} Cdyn;
```

```
void Cdyn_print(Cdyn *);
```

- and implement the function with above C prototype in Fortran
- → need read and/or write access to data allocated within the C-defined structure
- allocation is performed as described in Case 1

Case 3 (cont'd): Fortran implementation

Required type definition:

```
TYPE, BIND(C) :: cdyn
  INTEGER(c_int) :: len
  TYPE(c_ptr) :: f
end type cdyn
```

interoperates
with struct type `Cdyn`

Notes:

- note the **INTENT(IN)** for **this** (refers to association of **c_ptr**; the referenced data can be modified)
- **scenario 1** applies for **c_f_pointer** usage

Implementation:

```
SUBROUTINE cdyn_print(this) BIND(C,NAME='Cdyn_print')
  TYPE(cdyn), INTENT(IN) :: this
  REAL(c_float), POINTER :: cf(:)
  ! associate array pointer cf with this%f
  CALL c_f_pointer( this%f, cf, [this%len] )
  ! now do work with data pointed at by this%f
  WRITE(*,FMT=...) cf
END SUBROUTINE cdyn_print
```

See [examples/interop_c](#)

■ Procedure argument: a function pointer in C

- could have a fixed or variable interface
- example C prototype:

```
double integrate(double, double, void *,
                 double (*)(double, void *));
```

describes integrand
function

■ Matched by interoperable Fortran interface

```
REAL(c_double) FUNCTION integrate(a, b, par, fptr) bind(c)
  REAL(c_double), VALUE :: a, b
  TYPE(c_ptr), VALUE :: par
  TYPE(c_funptr), VALUE :: fptr
END FUNCTION
```

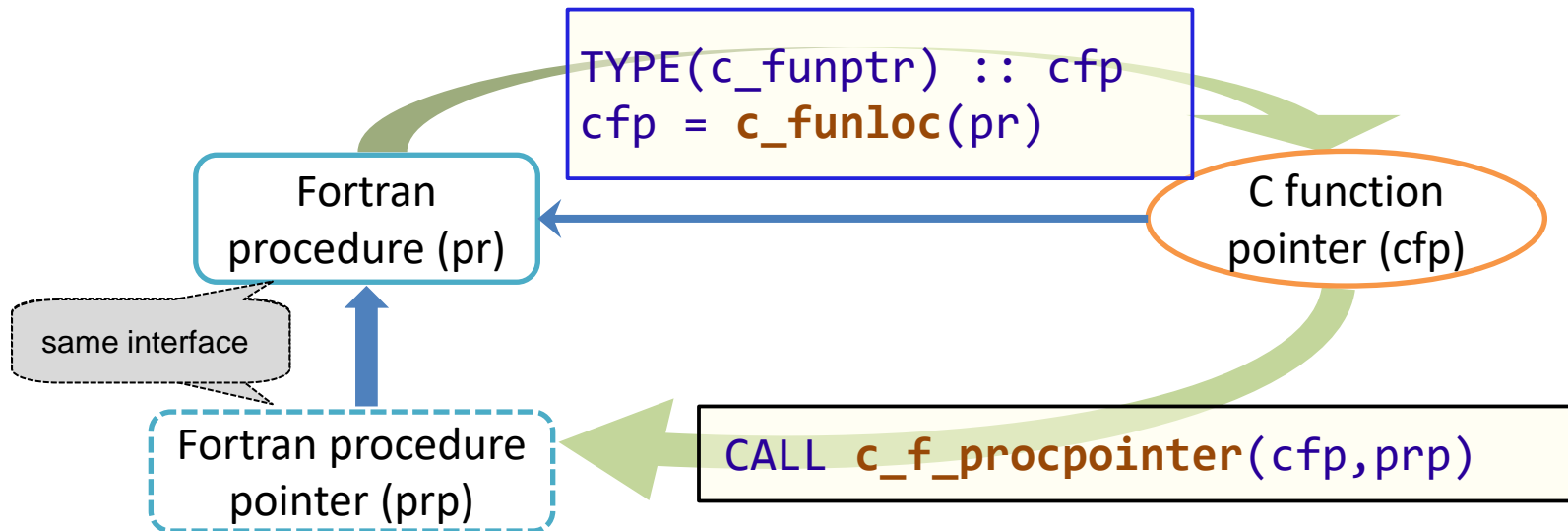
a C function pointer

■ Note:

- an interface with a Fortran procedure dummy argument is **not** interoperable (even if the dummy procedure has the BIND(C) attribute)

Setting up a mapping between a Fortran procedure and a C procedure pointer

Module ISO_C_BINDING provides module procedures



- input for `c_funloc` must be an **interoperable** Fortran procedure; can also be an associated procedure pointer
- pointer association (blue arrow) is set up as a result of their invocation (green arrows)

■ Assuming the function interface

note the consistent interface

```
REAL(c_double) FUNCTION f(x, par) bind(c)
  REAL(c_double), VALUE :: x
  TYPE(c_ptr), VALUE :: par
END FUNCTION
```

the invocation reads

```
TYPE(c_funptr) :: fp
fp = c_funloc(f)
res = integrate( a, b, par, fp )
```

or, more concisely

```
res = integrate( a, b, par, c_funloc(f) )
```

■ C function pointer used as type component

```
typedef struct {  
    double (*f)(double, void *);  
    void *par;  
} ParFun;
```

■ Matching type definition in Fortran:

- requires use of component of type `c_funptr`

```
TYPE, BIND(C) :: parfun  
    TYPE(c_funptr) :: f  
    TYPE(c_ptr) :: par  
END TYPE
```

Example:

```
TYPE(parfun) :: o_pf  
o_pf%f = c_funloc(my_function)  
o_pf%par = c_loc(...)
```

```
ParFun *o_pf;  
o_pf->f = my_function;  
o_pf->par = (void *) ...;
```

- where `my_function` should have the **same** interface in Fortran and C, respectively.

```
TYPE(parfun) :: o_pf  
TYPE(c_ptr) :: par  
PROCEDURE(my_function), POINTER :: pf  
  
: ! initialize o_pf, par within C or Fortran  
CALL c_f_procpointer(o_pf%f, pf)  
y = pf(2.0_dk, par)
```


■ Defining C code:

```
struct coord{  
    float xx, yy  
};  
struct coord csh;
```

- do not place in include file
- reference with `external` in other C source files

■ Mapping Fortran code

```
REAL(c_float) :: x, y  
COMMON /csh/ x, y  
BIND(C) :: /csh/
```

- **BIND statement** (possibly with a label) resolves to the same linker symbol as defined in C → **same memory address**
- memory layout may be different as for „traditional“ sequence association

■ Defining C code:

```
int ic;  
float Rpar[4];
```

- do not place in include file
- reference with `external` in other C source files

■ Mapping Fortran code:

```
MODULE mod_globals  
  USE, INTRINSIC :: iso_c_binding  
  
  INTEGER(c_int), BIND(C) :: ic  
  REAL(c_float) :: rpar(4)  
  BIND(C, NAME='Rpar') :: rpar  
END MODULE
```

- either attribute or statement form may be used

Global binding can be applied to objects of interoperable type and type parameters.

Variables with the `ALLOCATABLE/POINTER` attribute are not permitted in this context.

Enumeration

■ Set of integer constants

- only for interoperability with C

```
ENUM, BIND(C)
  ENUMERATOR :: red=4, blue=9
  ENUMERATOR :: yellow
END ENUM
```

- integer of same kind as used in C enum
- value of `yellow` is 10
- not hugely useful

■ **Preliminary specification (2012): „Mini-standard“ ISO/IEC TS 29113**

■ **Motivations:**

- enable a standard-conforming MPI (3.1) Fortran interface
- permit C programmers (limited) access to „complex“ Fortran objects

Area of semantics	within Fortran	within C
dummy argument POINTER or ALLOCATABLE	assumed shape/length or deferred shape/length	pointer to a descriptor
dummy argument	assumed rank	pointer to a descriptor
dummy argument	assumed type	either <code>void *</code> or pointer to a descriptor
dummy argument	OPTIONAL attribute no VALUE attribute permitted	use a NULL actual or check formal for being NULL
dummy argument of type <code>c_ptr</code> or <code>c_funptr</code>	non-interoperable data or procedure	handle only, no access to data or procedure
non-blocking procedures	ASYNCHRONOUS attribute	not applicable

Accessing Fortran infrastructure from C: the source file `ISO_Fortran_binding.h`

■ Example Fortran interface

```
SUBROUTINE process_array(a) BIND(C)
  REAL(c_float) :: a(:, :)
END SUBROUTINE
```

assumed
shape

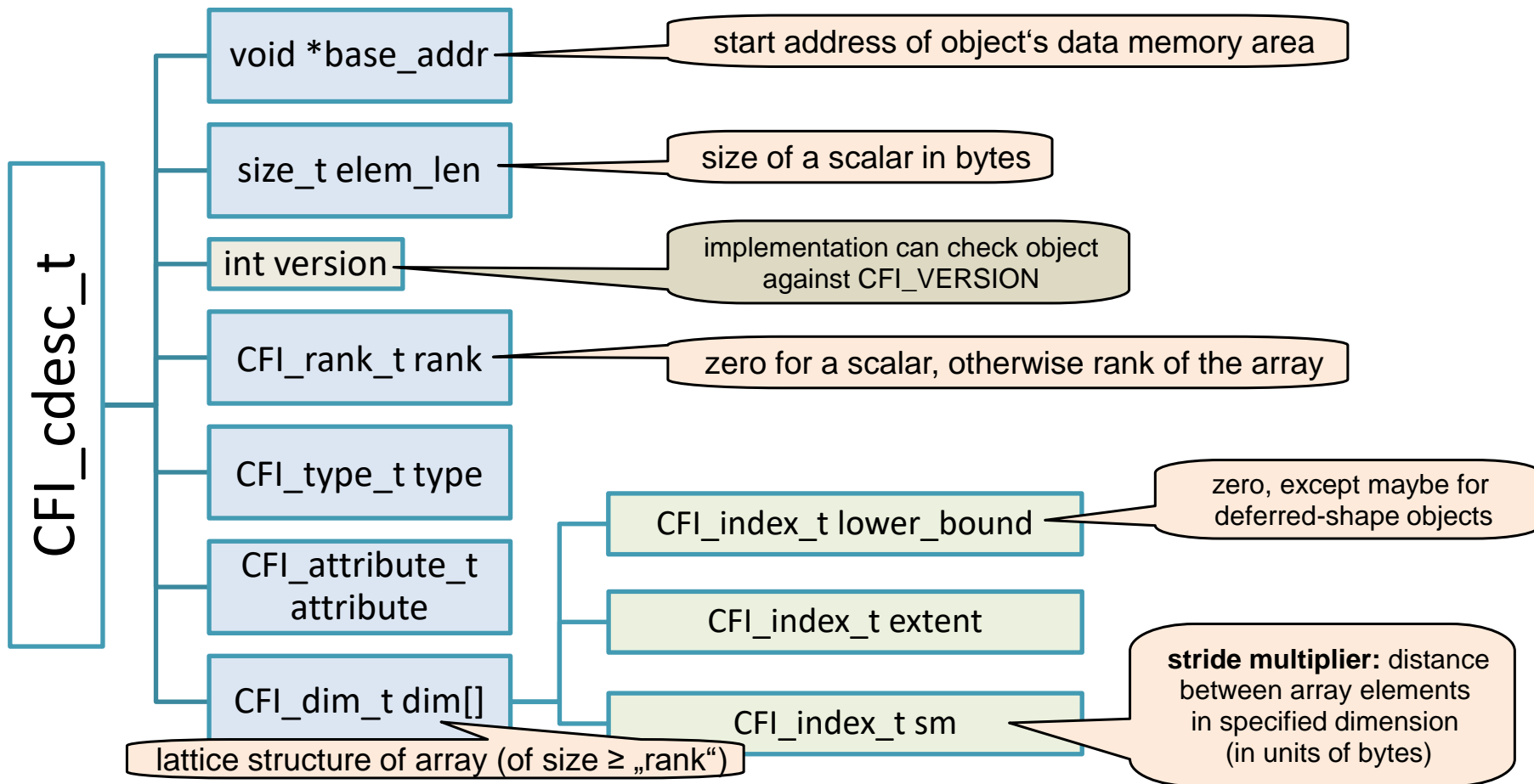
■ Matching C prototype

```
#include <ISO_Fortran_binding.h>
void process_array(CFI_cdesc_t *a);
```

- Implementation of procedure might be in C or in Fortran
- For an implementation in C, the header provides access to
 - type definition of descriptor (details upcoming ...)
 - macros for type codes, error states etc.
 - prototypes of library functions that generate or manipulate descriptors
- **Reserved** namespace: `CFI_`
- Within a single C source file,
 - binding is only possible to one given Fortran processor (no binary compatibility!)

■ Exposes internal structure of Fortran objects

- not meant for tampering  → please use API



■ Type code macros

- most commonly used:

Name	C type
CFI_type_int	int
CFI_type_long	long int
CFI_type_size_t	size_t
CFI_type_float	float
CFI_type_double	double
CFI_type_Bool	_Bool
CFI_type_char	char
CFI_type_cptr	void *
CFI_type_struct	Interoperable C structure
CFI_type_other (<0)	Not otherwise specified

typically, non-interoperable data

■ Attribute of dummy object

Name
CFI_attribute_allocatable
CFI_attribute_pointer
CFI_attribute_other

e.g., assumed shape or length

- **Beware:** attribute value of actual must match up **exactly** with that of dummy (different from Fortran) → may need to create descriptor copies

■ Fortran reference loop within `process_array()`:

```
DO k=1, ubound(a, 2)
  DO i=1, ubound(a, 1)
    ... = a(i, k) * ...
  END DO
END DO
```

Remember: „a“ represents
a rank-2 array of
assumed shape

■ C implementation variant 1:

ordering of dimensions
as in Fortran

```
for (k = 0; k < a->dim[1].extent; k++) {
  for (i = 0; i < a->dim[0].extent; i++) {
    CFI_index_t subscripts[2] = { i, k };
    ... = *((float *) CFI_address( a, subscripts )) * ...;
  }
}
```

- `CFI_address()` returns `(void *)` address of array element indexed by specified (valid!) subscripts
- `dim[].lower_bound` will be needed for pointer/allocatable objects

■ C implementation variant 2:

- start out from beginning of array

```
char *a_ptr = (char *) a->base_addr;
```

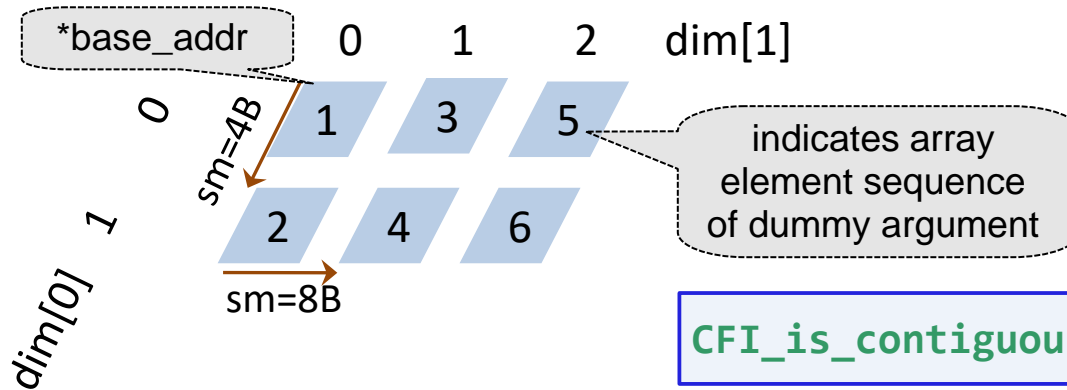
and use pointer arithmetic to process it:

```
char *a_aux;  
for (k = 0; k < a->dim[1].extent; k++) {  
    a_aux = a_ptr;  
    for (i = 0; i < a->dim[0].extent; i++) {  
        ... = *((float *) a_ptr) * ...;  
        a_ptr += a->dim[0].sm;  
    }  
    a_ptr = a_aux + a->dim[1].sm;  
}
```

pointer to Fortran
element a(i,k)

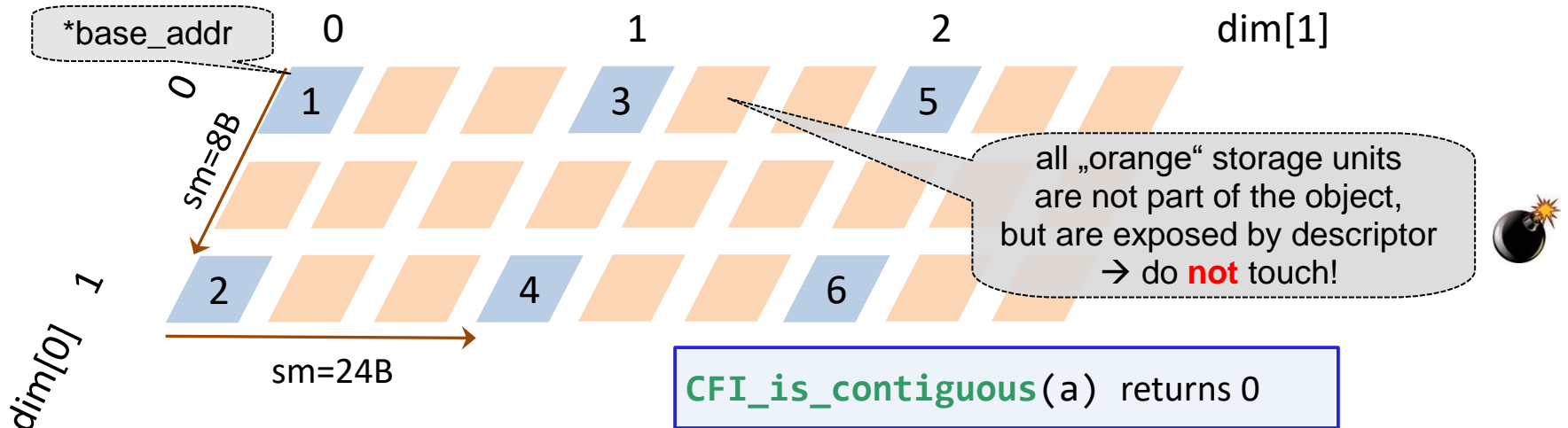
- non-contiguous arrays require use of stride multipliers (next slide illustrates why)
- stride multipliers in general may not be an integer multiple of the element size → always process in units of bytes

Actual argument is a complete array (0:1,3)



`CFI_is_contiguous(a)` returns 1

Actual argument is an array section (0::2,1::3) of (0:2,9)



`CFI_is_contiguous(a)` returns 0

- May be necessary to invoke a Fortran procedure from C

- **Step 1: create a descriptor**

```
CFI_CDESC_T(2) A;
```

use macro to establish needed storage;
maximum rank must be specified as parameter

```
CFI_cdesc_t *a = (CFI_cdesc_t *) &A;
```

cast to properly
typed pointer

- **Step 2: establish object's properties**

- prototype of function to be used for this is

```
int CFI_establish (  
    CFI_cdesc_t *dv,  
    void *base_addr,  
    CFI_attribute_t attribute,  
    CFI_type_t type,  
    size_t elem_len,  
    CFI_rank_t rank,  
    const CFI_index_t extents[]  
);
```

- many usage patterns
- if fully defined, result is always a contiguous object
- function result is an error indicator (CFI_SUCCESS → OK)

Macro name	Explanation of error
CFI_SUCCESS	No error detected.
CFI_ERROR_BASE_ADDR_NULL	The base address member of a C descriptor is a null pointer in a context that requires a non-null pointer value.
CFI_ERROR_BASE_ADDR_NOT_NULL	The base address member of a C descriptor is not a null pointer in a context that requires a null pointer value.
CFI_INVALID_ELEM_LEN	The value supplied for the element length member of a C descriptor is not valid.
CFI_INVALID_RANK	The value supplied for the rank member of a C descriptor is not valid.
CFI_INVALID_TYPE	The value supplied for the type member of a C descriptor is not valid.
CFI_INVALID_ATTRIBUTE	The value supplied for the attribute member of a C descriptor is not valid.
CFI_INVALID_EXTENT	The value supplied for the extent member of a CFI_dim_t structure is not valid.
CFI_INVALID_DESCRIPTOR	A general error condition for C descriptors.
CFI_ERROR_MEM_ALLOCATION	Memory allocation failed.
CFI_ERROR_OUT_OF_BOUNDS	A reference is out of bounds.

Example: create rank 2 assumed shape array

```
#define DIM1 56
#define DIM2 123

CFI_CDESC_T(2) A;          /* 2 is the minimum value needed */
CFI_cdesc_t *a = (CFI_cdesc_t *) &A;
CFI_index_t extents[2] = { DIM1, DIM2 };
                          /* shape of rank 2 array */

float *a_ptr = (float *) malloc(DIM1*DIM2*sizeof(float));
                          /* heap allocation within C */
...                          /* initialize values of *a_ptr */
CFI_establish( a, (void *) a_ptr,
               CFI_attribute_other,
               CFI_type_float,
               0,          /* elem_len is ignored here */
               2,          /* rank as declared in Fortran */
               extents );
                          /* have a fully defined object now */
process_array(a);

free(a_ptr);              /* object becomes invalid */
```

- Typically only needed if Fortran API defines a „factory“:

```
TYPE, BIND(C) :: qbody
  REAL(c_float) :: mass
  REAL(c_float) :: position(3)
END TYPE
INTERFACE
  SUBROUTINE qbody_factory(this, fname) BIND(C)
    TYPE(qbody), ALLOCATABLE, INTENT(OUT) :: this(:,,:)
    CHARACTER(c_char, LEN=*), INTENT(IN) :: fname
  END SUBROUTINE
END INTERFACE
```

```
typedef struct {
  float mass;
  float position[3];
} qbody;
```

matching C
struct definition

- matching C prototype:

```
void qbody_factory(CFI_cdesc_t *, CFI_cdesc_t *)
```

descriptor corresponds to
assumed length character

Example: create an allocatable entity and populate it

```
char fname_ptr[] = "InFrontOfMyHouse.dat";

CFI_cdesc_t *pavement =
    (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
CFI_cdesc_t *fname =
    (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(0)));
CFI_establish( pavement, NULL, CFI_attribute_allocatable,
    CFI_type_struct,
    sizeof(qbody), /* derived type object size */
    2, NULL );
CFI_establish( fname, fname_ptr, CFI_attribute_other,
    CFI_type_char,
    strlen(fname_ptr), /* a char has one byte */
    0, NULL );
qbody_factory ( pavement, fname ); /* object is created */
... /* process pavement */
CFI_deallocate( pavement );
free(pavement); free(fname);
```

must start out unallocated

shape is deferred

no auto-deallocation of objects allocated in C

```
void qbody_factory(CFI_cdesc_t *this, CFI_cdesc_t *fname_str) {
    char *fname = (char *) fname_str->base_addr;
    CFI_index_t lowerbds[2], upperbds[2];
    ...      /* open file *fname and read in bounds information */
    if (this->base_addr != NULL) CFI_deallocate(this);
        /* emulate INTENT(OUT) semantics for C-to-C calls */
    CFI_allocate(this, lowerbds, upperbds, 0);
        /* object is now allocated */
    ...      /* read object data from file *fname */
}
```

may want to do run time checks

■ Feasible because of supplied function `CFI_allocate()`:

- last argument is an element length, which is ignored unless the type member is `CFI_type_char`. In the latter case, its value becomes the element length of the allocated deferred-length (!) string.

■ Anonymous target

- create descriptor with CFI_attribute_pointer, then apply CFI_allocate()/CFI_deallocate()

■ Point at an existing target

```
REAL(c_float), TARGET :: t(:)
REAL(c_float), POINTER :: p(:)
p(3:) => t
```

```
CFI_index_t lower_bounds[1] = { 3 };

status = CFI_setpointer( p, t,
                        lower_bounds );
```

- **t** must describe a fully valid object
- **p** must be an established descriptor with CFI_attribute_pointer and for the same type as **t**.



Beware: No compile-time type safety is provided.

Certain inconsistencies may be diagnosed at run time

→ check return value of CFI_setpointer()

■ Assumption:

- arr describes an assumed-shape rank 3 array

■ Create a descriptor for the section arr(3:,4,::2)

```
CFI_cdesc_t *section =
    (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
CFI_index_t lower_bounds[3] = { 2, 3, 0 };
CFI_index_t upper_bounds[3] =
    { arr->dim[0].extent - 1, 3, arr->dim[2].extent - 1 };
CFI_index_t strides[3] = { 1, 0, 2 };
```

zero stride indicates a subscript.
For this dimension, lower and upper bounds must be equal.

```
CFI_establish( section, NULL, CFI_attribute_other,
    arr->type, arr->elem_len, 2, NULL );
    /* section here is an undefined object */
CFI_section( section, arr, lower_bounds, upper_bounds, strides );
    /* now, section is defined */
```

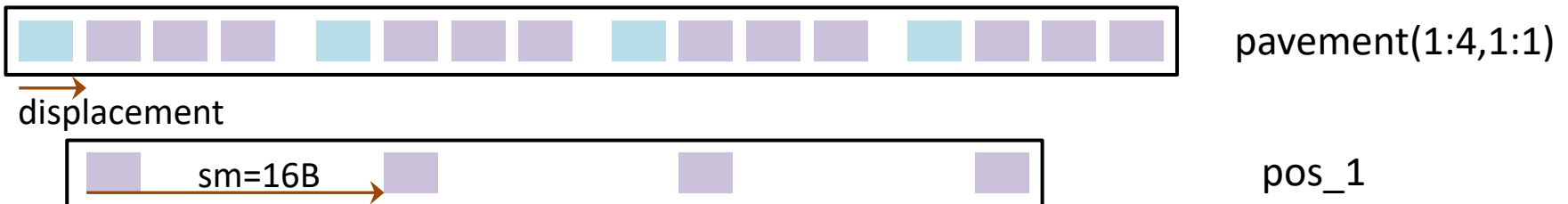
■ Type component selection

- pavement(:)%position(1) from the type(qbody) object pavement
- a rank-2 array of intrinsic type real(c_float)

```
CFI_cdesc_t *pos_1 =
    (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
size_t elem_len = 0;
CFI_establish( pos_1, NULL, CFI_attribute_other,
    CFI_type_float, elem_len, 2, NULL );
    /* pos_1 here is an undefined object */

size_t displacement = offsetof(qbody, position[0]);
CFI_select_part( pos_1, pavement, displacement, elem_len );
    /* now, pos_1 is defined */
```

ignored here
(only relevant for strings)



■ Enables invocation of appropriately declared object

```
SUBROUTINE process_allranks(ar, ...)  
  REAL :: ar(..)  
  ...  
  WRITE(*,*) rank(ar)  
END SUBROUTINE
```

ar cannot (currently) be referenced or defined within Fortran.

However, some intrinsics can be invoked.

with arrays of **any** rank, or even a scalar:

```
REAL :: xs, x1(4), x2(ndim, 4)  
  
CALL process_allranks(xs, ...)    scalar  
CALL process_allranks(x1, ...)   rank 1  
CALL process_allranks(x2, ...)   rank 2
```

avoid need for writing many specifics for a named interface

■ Assuming the procedure interface is made **BIND(C)**:

- descriptor always contains well-defined rank information

```
void process_allranks(CFI_cdesc_t *ar, ...) {  
    switch( ar->rank )  
    case 1:  
        ... /* process single loop nest */  
    case 2:  
        ... /* process two nested loops */  
    default:  
        printf("unsupported rank value\n");  
        exit(1);  
    }  
}
```

as seen earlier

- deep loop nests can be avoided for contiguous objects, but the latter is not assured

■ Special case:

- size of (contiguous) assumed-size object is not known

```
REAL :: x2(ndim, *)  
CALL process_allranks(x2, ..., ntot)
```

should specify size
separately

■ A descriptor with following properties is constructed:

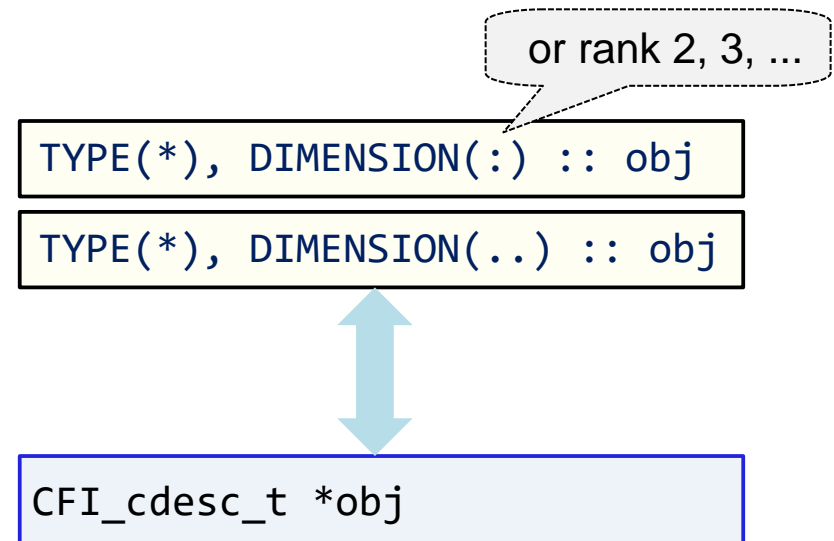
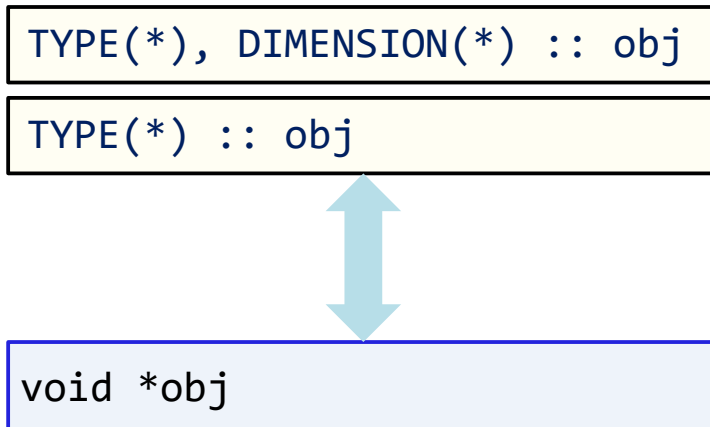
- `SIZE(ar,DIM=RANK(ar))` has the value **-1**
- `UBOUND(ar,DIM=RANK(ar))` has the value `UBOUND(ar,DIM=RANK(ar)) - 2`

■ Declaration with TYPE(*)

- an unlimited polymorphic object → actual argument may be of any type
- dynamic type cannot change → no POINTER or ALLOCATABLE attribute is permitted

■ Corresponding object in interoperating C call:

- two variants are possible



■ C prototype as specified in the MPI standard

```
int MPI_Send( const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm );
```

■ Matching Fortran interface:

```
INTEGER(c_int) FUNCTION C_MPI_Send( buf, count, datatype, dest, &
                                   tag, comm ) BIND(C, name="MPI_Send")
  TYPE(*), DIMENSION(*), INTENT(IN) :: buf
  INTEGER(c_int), VALUE :: count, dest, tag
  TYPE(MPI_Datatype), VALUE :: datatype
  TYPE(MPI_Comm), VALUE :: comm
END FUNCTION C_MPI_Send
```

size of memory area
specified by count
and datatype

- assumes interoperable types MPI_Datatype etc.
- actual argument may be array or scalar
- non-contiguous actuals are compactified

array temps are a problem
for non-blocking calls


```
SUBROUTINE MPI_Send( buf, count, datatype, dest, &
                    tag, comm, ierror )
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN):: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
END FUNCTION MPI_Send
```

■ Invocation of MPI_Send

- now possible also with **array section** actual arguments without need for copy-in/out

■ Could add BIND(C) to the interface for a C implementation

- assuming int matches default Fortran integer
- the MPI standard doesn't do this, though

```
void mpi_send( CFI_cdesc_t *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, int *ierror ) {
    int ierror_local;
    MPI_Datatype disc_type;
    if ( CFI_is_contiguous( buf ) ) {
        ierror_local = MPI_Send( buf->base_addr, count, datatype,
                                dest, tag, comm );
    } else {
        ... /* use descriptor information to construct disc_type
            from datatype (e.g. via MPI_Type_create_subarray) */

        ierror_local = MPI_Send( buf->base_addr, count, disc_type,
                                dest, tag, comm );
        ... /* clean up disc_type */
    }
    if (ierror != NULL) *ierror = ierror_local;
}
```

Automatized translation of C include files to Fortran interface modules

- **Requires a specialized tool**
 - for example, Garnet Liu's LLVM-based tool, see <https://github.com/Kaiveria/h2m-Autofortran-Tool>
- **C include files can have stuff inside that is not covered by interoperability**
 - only a subset can be translated
- **Topic goes beyond the scope of this course**

■ Interoperation with C++

- no direct interoperation with C++-specific features is possible
- you need to write C-like bridge code
- declare C-style functions „extern C“ in your C++ sources
- explicit linkage of C++ libraries will be needed if the Fortran compiler driver is used for linking

■ Vararg interfaces

- are not interoperable with any Fortran interface
- you need to write glue code in C

Shared Libraries and Plug-ins

■ Executable code in library

- is shared between all programs linked against the library (instead of residing in the executable)
- this does not apply to data entities

■ Advantages:

- save memory space
- save on access latency
- bug fixes in library code do not require relinking the application

■ Disadvantages:

- higher complexity in handling the build and packaging of applications
- (need to distribute shared libraries together with the linked application)
- not supported (in analogous manner) on all operating environments
- (will focus on ELF-based Linux in this talk)
- special compilation procedure is required for library code

ELF → executable and linkable format
see http://en.wikipedia.org/wiki/Executable_and_Linkable_Format for details

■ Causes of incompatibility

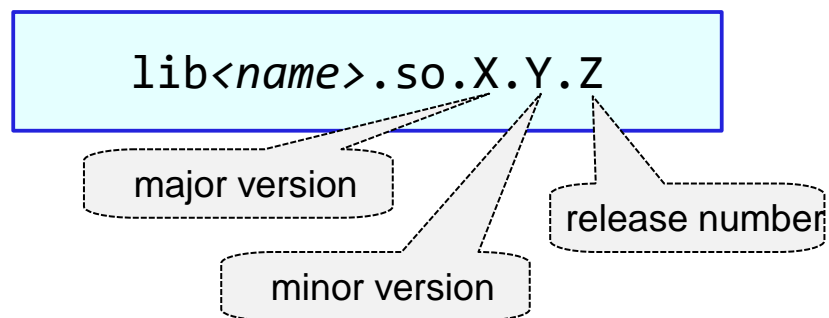
- change function interface
- remove function interface
- (adding a new function is no problem)
- change in type definitions or data items
- (exception: storage association is preserved)
- changes in function behaviour

■ If one of these happens,

- a mechanism is available to indicate the library is not compatible

■ Concept of **soname**

- naming scheme for shared libraries



- soname will typically be

`lib<name>.so.X`

- and the latest version of the library with the same soname should be picked up at linkage

Example (1): building a library

Assume you have

- a source file `mylib.f90`

separate steps

Implementation-dependent options

Compiler	Compilation	Linkage
Intel ifort	<code>-fPIC</code>	<code>-shared -Wl,-soname lib<name>.so.X</code>
Gfortran	<code>-fPIC</code>	<code>-shared -Wl,-soname=lib<name>.so.X</code>
PGI pgf90	<code>-fPIC</code>	<code>-shared -Wl,-soname=lib<name>.so.X</code>
NAG nagfor	<code>-PIC</code>	<code>-shared -Wl,-soname=lib<name>.so.X</code>
IBM xlf	<code>-G -qp-pic=big</code>	<code>-qmkschroobj</code>
Cray ftn	?	?

Example (Intel compiler):

```
ifort -c -fPIC mylib.f90
ifort -o libmylib.so.1.0.0 -shared -Wl,-soname libmylib.so.1 mylib.o
```

Add symbolic links (Linux)

```
ln -s libmylib.so.1.0.0 libmylib.so.1
ln -s libmylib.so.1 libmylib.so
```


Linux linkage:

- specify **directory** where shared library resides
- specify **shorthand** for library name

```
ifort -o myprog.exe myprog.f90 -L../lib -lmylib
```

- **note:** if both a static and a shared library are found, the shared library will be used by default
- there usually exist compiler switches which enforce static linking

Execute binary

- set **library path**
- **execute** as usual

```
export LD_LIBRARY_PATH=\  
$HOME/lib:$LD_LIBRARY_PATH  
./myprog.exe
```

libmylib.so lives there

- **note:** `/etc/ld.so.conf` contains library paths which are always searched
- there usually exist possibilities to hard-code the library path into the executable

don't need to set LD_LIBRARY_PATH in these two cases

■ The **-Wl**, option can be used to pass options to the linker

■ **Example 1:**

- want to specify that a certain library **-lspecial** should be linked statically, others dynamically
- this is not uniquely resolvable from the library specification if both static and dynamic versions exist!

```
ifort -o myprog.exe myprog.f90 -Wl,-static -L/special_path \  
-lspecial -Wl,-dy -L../lib -lmylib
```

■ **Example 2: hard-code path into binary**

```
ifort -o myprog.exe myprog.f90 -Wl,-rpath -L../lib \  
-L../lib -lmylib
```

- avoids the need to set LD_LIBRARY_PATH before execution

■ Supported by C library:

- open a shared library at run time
- extract a symbol (function pointer)
- execute function
- close shared library

man 3p dlopen / dlsym / dlclose

■ From Fortran

- usable via C interoperability and pointers to procedures
- implement plug-ins

■ Small Fortran module `dlfcn`

- type definition `dlfcn_handle`
- procedures `dlfcn_open()`, `dlfcn_symbol()`, `dlfcn_close()`
- **Note:** the result of `dlfcn_symbol()` is of type `c_funptr` to enable conversion to an explicit interface procedure pointer
- constants required for `dlfcn_open()` mode

```
USE dlfcn
IMPLICIT NONE
ABSTRACT INTERFACE
  SUBROUTINE set(i) BIND(C)
    INTEGER, INTENT(INOUT) :: i
  END SUBROUTINE SET
END INTERFACE
INTEGER :: i, istat
TYPE(dlfcn_handle) :: h
TYPE(c_funptr) :: cp
PROCEDURE(set), POINTER :: fp

h = dlfcn_open('./libset1.so', &
              RTLD_NOW)
cp = dlfcn_symbol(h, 'set_val')
CALL c_f_procpointer(cp, fp)
i = 1
CALL fp(i)
istat = dlfcn_close(h)
```

procedure
name

Shared library `libset1.so`:

- BIND(C) procedure

Module procedure:

- explicit name mangling needed

```
h = dlfcn_open('./libset2.so', &
              RTLD_NOW)
! at most one line valid
cp = dlfcn_symbol(h, &
                 '__s_MOD_set_val')
cp = dlfcn_symbol(h, &
                 's_mp_set_val_')
cp = dlfcn_symbol(h, &
                 '__s_NMOD_set_val')
cp = dlfcn_symbol(h, &
                 's_MP_set_val')
```

gfortran
ifort
xlf
NAG, g95

```
CALL c_f_procpointer(cp, fp)
i = 1
CALL fp(i)
istat = dlfcn_close(h)
```

OK with
F18

```
nm libset2.so | grep -i set_val
```

This concludes the LRZ part of this course

Following now: Exercise session 5