Fortran interoperability, in practice

Connecting Fortran to other languages with `iso_c_binding`

Why interoperability?

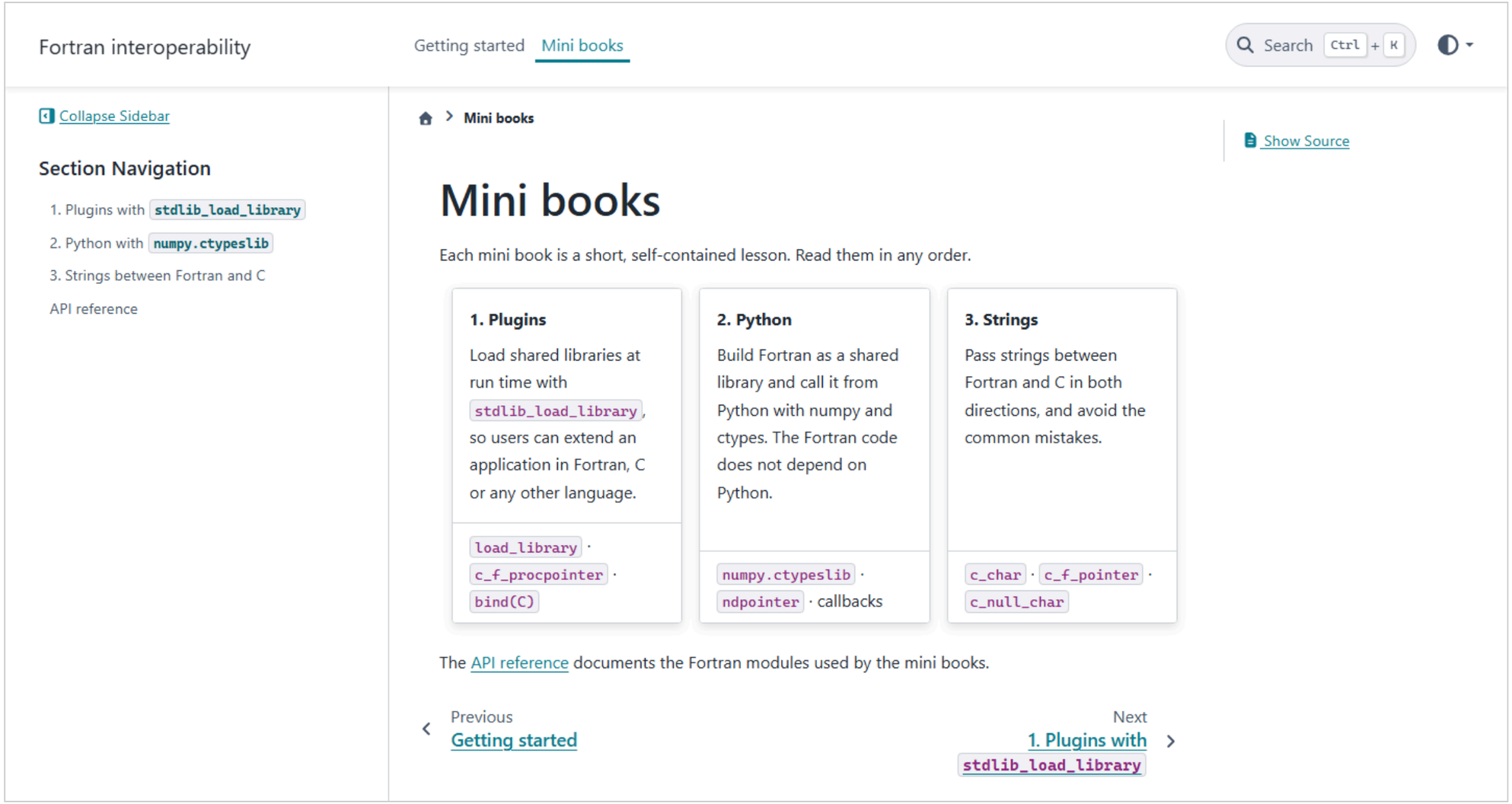
- Reuse Fortran code from Python, C, C++, Julia, R…
- Let users extend a Fortran application without rebuilding it
- Call existing C libraries from Fortran

All through one door: **the C calling convention**, which every language speaks.

Outline

0. **Basics**: `iso_c_binding` in one slide
1. **Plugins**: a contract, plugins in Fortran and C, loading them at run time, mixing compilers, Fortran 2018 descriptors
2. **Python**: a C interface around plain Fortran, numpy arrays, callbacks, strings
3. **Strings**: Fortran to C, C to Fortran, buffers, a non-conforming idiom

The mini books, online



Each part of this talk is a mini book, with the full code and commands.

`iso_c_binding` in one slide

Fortran	C
`integer(c_int), value :: n`	`int n`
`real(c_double) :: x(n)`	`double *x`
`character(kind=c_char) :: s(*)`	`char *s`
`type(c_ptr), value :: p`	`void *p`
`type(c_funptr), value :: f`	`double (*f)(double)`
`function f(...) bind(C, name="f")`	`double f(...)`

`bind(C, name="...")` fixes the symbol name, whatever the compiler.

Without `value`, Fortran passes the address: `type(c_ptr) :: p` is `void **p`.

The three mini books

1. **Plugins**: load shared libraries at run time with `stdlib_load_library`
2. **Python**: call Fortran from Python with `numpy.ctypeslib`
3. **Strings**: pass strings between Fortran and C

Each one is a small fpm project.

Setup

```
conda create -n interop -c conda-forge fpm gfortran numpy
conda activate interop
fpm build
fpm run          # lists the commands of each mini book
```

Add `--compiler ifx` to any fpm command for Intel Fortran.

1. Plugins

Extending an application at run time with `stdlib_load_library`

The goal

Users change what an application does **without rebuilding it**: they write a *plugin*, a shared library the application loads at run time.

Our application solves

$$\frac{dy}{dt} = f(t, y), \qquad y(0) = 1$$

and the plugin supplies f.

Same design: material laws, boundary conditions, optional back-ends…

The pieces

```
src/plugin_api.f90       the contract, in Fortran
include/plugin_api.h     the same contract, in C
example/plugin_host/     the application
plugins/decay/           a plugin in Fortran
plugins/logistic/        a plugin in C
```

Each plugin is its own fpm project: built separately, by anyone, with any compiler.

The contract

C prototype	Required
`int plugin_api_version(void)`	yes
`const char *plugin_name(void)`	yes
`double plugin_rhs(double t, double y)`	yes
`double plugin_exact(double t)`	no

- a **version number**, checked first
- **only C types** cross the boundary
- **clear ownership**: the plugin keeps its name string alive

The contract in Fortran

```
integer(c_int), parameter :: API_VERSION = 1

abstract interface
    !> ``int plugin_api_version(void)``
    function api_version_t() bind(C) result(version)
        import :: c_int
        integer(c_int) :: version
    end function api_version_t

    !> ``const char *plugin_name(void)``: a NUL-terminated string that the
    !> plugin owns and keeps alive while it is loaded.
    function name_t() bind(C) result(name)
        import :: c_ptr
        type(c_ptr) :: name
    end function name_t

    !> ``double plugin_rhs(double t, double y)``: :math:`f(t, y)`
    function rhs_t(t, y) bind(C) result(dydt)
        import :: c_double
        real(c_double), value :: t, y
        real(c_double) :: dydt
    end function rhs_t
```

A plugin in Fortran

```
function plugin_name() bind(C, name="plugin_name") result(name)
    !DEC$ ATTRIBUTES DLLEXPORT :: plugin_name
    type(c_ptr) :: name
    name = c_loc(label_storage)
end function plugin_name

function plugin_rhs(t, y) bind(C, name="plugin_rhs") result(dydt)
    !DEC$ ATTRIBUTES DLLEXPORT :: plugin_rhs
    real(c_double), value :: t, y
    real(c_double) :: dydt
    dydt = -k*y
end function plugin_rhs
```

`!DEC$ ATTRIBUTES DLLEXPORT`: needed by ifx on Windows, a comment elsewhere.

A plugin in C

```c
#include "plugin_api.h"

static const double r = 1.0, K = 10.0;

int plugin_api_version(void) { return PLUGIN_API_VERSION; }

/* A string literal lives as long as the library: safe to hand out. */
const char *plugin_name(void) { return "logistic growth (C)"; }

double plugin_rhs(double t, double y)
{
    (void)t;
    return r * y * (1.0 - y / K);
}
```

The host cannot tell the difference.

Build and deploy the plugins

```
name = "decay"
version = "0.1.0"

[library]
type = "shared"

[install]
library = true
```

```
cd plugins/decay    && fpm install --prefix ../../build/plugins --libdir .
cd plugins/logistic && fpm install --prefix ../../build/plugins --libdir .
```

Load the library, check the version

```
use stdlib_load_library, only: shared_library_type, load_library
use stdlib_error, only: state_type
```

```
lib = load_library(name, path=dir, err=err)
if (err%error()) then
    print '(a)', err%print()
    error stop 1
end if
print '(2a)', "loaded : ", lib%filename()

! 2. Check the contract before calling anything else
call c_f_procpointer(lib%symbol("plugin_api_version"), get_version)
if (get_version() /= API_VERSION) then
    print '(a,i0,a,i0)', "plugin API version ", get_version(), ", expected ", API_VERSION
    error stop 1
end if
```

`name` and `dir` come from the command line (`plugin_host decay build/plugins`). A bare name is decorated for the platform: `"decay"` finds `libdecay.so`, `libdecay.dylib`, `decay.dll` or `libdecay.dll`.

Symbols become procedure pointers

```
! 3. Required symbols: without `err`, a missing one stops the program
call c_f_procpointer(lib%symbol("plugin_name"), plugin_name)
call c_f_procpointer(lib%symbol("plugin_rhs"), rhs)

! 4. Optional symbol: look before you leap
if (lib%has_symbol("plugin_exact")) then
    call c_f_procpointer(lib%symbol("plugin_exact"), exact)
end if

print '(2a)', "plugin : ", c_to_f(plugin_name())
```

`lib%symbol` gives a `c_funptr`; `c_f_procpointer` gives it a Fortran interface.

Use it, then close it

```
do i = 1, nsteps
    k1 = rhs(t, y)
    k2 = rhs(t + dt, y + dt*k1)
    y = y + dt*(k1 + k2)/2
    t = i*dt
    call report(t, y)
end do

! 6. Unloading is explicit. Procedure pointers into the library are now invalid
call lib%close()
```

Closing invalidates every procedure pointer into the library.

Run it

```
$ fpm run --example plugin_host -- decay build/plugins
plugin : exponential decay (Fortran)
t =  0.0   y =  1.000000   error =  0.00E+00
...
t =  2.0   y =  0.137448   error =  2.11E-03

$ fpm run --example plugin_host -- logistic build/plugins
plugin : logistic growth (C)
t =  0.0   y =  1.000000
...
t =  2.0   y =  4.494506
```

The optional `plugin_exact` at work: the Fortran plugin has it, so the host prints the error; the C plugin does not, so it prints none.

Mixing compilers

Host built with gfortran, plugins with ifx, or the reverse: same results.

Each library brings its own Fortran runtime:

- no I/O units, no `stop` in a plugin
- no derived types other than `bind(C)` ones

Fortran 2018 descriptors

`x(:)`, `character(*)`, `allocatable`, `pointer`, `x(..)` are interoperable through `CFI_cdesc_t`…
whose **layout is compiler-specific**:

`CFI_cdesc_t`	**gfortran**	**ifx**	**flang**
size, rank 1	48	96	48
`CFI_type_double`	2051	6	28

Caller → library	**`x(:)`**	**`optional`**
same compiler	correct	correct
ifx → gfortran	wrong, silently	correct
gfortran → ifx	crash	correct

Across compilers or languages: no descriptors. `optional` (a NULL pointer) is fine.

Pitfalls

- No finalizer: `call lib%close()` is explicit
- Linux with glibc < 2.34: link the host with `--link-flag -ldl`
- flang 22 on Windows: DLLs export no symbols

2. Python

Calling Fortran from Python with `numpy.ctypeslib`

The goal

Use Fortran from Python **without tying Fortran to Python**:

- a plain shared library with a C interface
- no Python headers, no wrapper generator, no extension module
- the same library also serves C, C++, Julia, R…

Three layers

Layer	Knows about	Job
`numerics.f90`	Fortran	the work
`numerics_capi.f90`	Fortran and C	convert, delegate
`numerics.py`	C and Python	prototypes, checks

```
python/
├── fpm.toml              a shared library project
├── src/numerics.f90      plain Fortran
├── src/numerics_capi.f90 bind(C) wrappers
├── numerics.py           the Python interface
└── demo.py
```

The core stays plain Fortran

```
pure subroutine mean_std(x, mean, std, stat)
    real(dp), intent(in) :: x(:)
    real(dp), intent(out) :: mean, std
    integer, intent(out) :: stat

    mean = 0
    std = 0
    if (size(x) < 2) then
        stat = 1
        return
    end if
    mean = sum(x)/size(x)
    std = sqrt(sum((x - mean)**2)/(size(x) - 1))
    stat = 0
end subroutine mean_std
```

Never `stop` in a library: it would kill the Python interpreter.

A C interface around it

```
function num_mean_std(n, x, mean, std) bind(C, name="num_mean_std") result(stat)
    !DEC$ ATTRIBUTES DLLEXPORT :: num_mean_std
    integer(c_int), value :: n
    real(c_double), intent(in) :: x(n)
    real(c_double), intent(out) :: mean, std
    integer(c_int) :: stat

    integer :: istat

    call mean_std(x, mean, std, istat)
    stat = istat
end function num_mean_std
```

Scalars by `value`, arrays with their sizes, outputs by reference.

2-D arrays: memory order

```
subroutine num_scale_columns(m, n, a, s) bind(C, name="num_scale_columns")
    !DEC$ ATTRIBUTES DLLEXPORT :: num_scale_columns
    integer(c_int), value :: m, n
    real(c_double), intent(inout) :: a(m, n)
    real(c_double), intent(in) :: s(n)

    call scale_columns(a, s)
end subroutine num_scale_columns
```

- Fortran: columns are contiguous (column-major)
- numpy: rows are contiguous by default (C order)
- `np.asfortranarray(a)` makes a Fortran-ordered copy

Build

```
cd python
fpm install --prefix .          # -> lib/libnumerics.so / .dll / .dylib
```

The project has **no dependencies**: fpm would build each one as another shared library for Python to find.

Declare the C prototypes

```
_lib = np.ctypeslib.load_library("libnumerics", os.path.join(_here, "lib"))

# -- C prototypes --------------------------------------------------------
# ndpointer checks dtype, rank and memory layout before Fortran sees the data.
_double_1d = ndpointer(dtype=np.float64, ndim=1, flags="C_CONTIGUOUS")
_double_2d_f = ndpointer(dtype=np.float64, ndim=2, flags="F_CONTIGUOUS")
_callback_t = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)

_lib.num_version.argtypes = []
_lib.num_version.restype = ctypes.c_char_p

_lib.num_mean_std.argtypes = [ctypes.c_int, _double_1d,
                              ctypes.POINTER(ctypes.c_double),
                              ctypes.POINTER(ctypes.c_double)]
_lib.num_mean_std.restype = ctypes.c_int

_lib.num_scale_columns.argtypes = [ctypes.c_int, ctypes.c_int, _double_2d_f, _double_1d]
_lib.num_scale_columns.restype = None
```

`ndpointer` checks dtype, rank and order **before** Fortran sees the data.

A Pythonic wrapper

```
def mean_std(x):
    """Mean and sample standard deviation of a 1-D array."""
    x = np.ascontiguousarray(x, dtype=np.float64)
    mean, std = ctypes.c_double(), ctypes.c_double()
    stat = _lib.num_mean_std(x.size, x, ctypes.byref(mean), ctypes.byref(std))
    if stat != 0:
        raise ValueError("mean_std needs at least two values")
    return mean.value, std.value
```

numpy arrays in, tuples and exceptions out.

Run it

```
python demo.py
```

```
library    : numerics 0.1.0
mean, std  : (2.5, 1.2909944487358056)
error      : mean_std needs at least two values
scaled     :
 [[  1.  20. 300.]
 [  4.  50. 600.]]
error      : argument 3: TypeError: array must have flags ['F_CONTIGUOUS']
integral   : 0.33333349999999995
greeting   : Hello Zoë, from Fortran!
```

Callbacks: Python called from Fortran

```
function num_trapezoid(f, a, b, n) bind(C, name="num_trapezoid") result(integral)
    !DEC$ ATTRIBUTES DLLEXPORT :: num_trapezoid
    type(c_funptr), value :: f
    real(c_double), value :: a, b
    integer(c_int), value :: n
    real(c_double) :: integral

    call c_f_procpointer(f, callback)
    integral = trapezoid(call_callback, a, b, int(n))
    callback => null()
end function num_trapezoid

!> Fortran-interface adapter around the C function pointer
function call_callback(x) result(y)
    real(dp), intent(in) :: x
    real(dp) :: y
    y = callback(x)
end function call_callback
```

```
callback = _callback_t(f)
return _lib.num_trapezoid(callback, a, b, n)
```

Strings

```
def greeting(name):
    """A greeting for ``name``, built in Fortran."""
    name = name.encode("utf-8")
    buf = ctypes.create_string_buffer(16)          # deliberately small
    needed = _lib.num_greeting(name, len(name), buf, len(buf))
    if needed >= len(buf):                         # truncated: once more,
        buf = ctypes.create_string_buffer(needed + 1)  # with room for the NUL
        _lib.num_greeting(name, len(name), buf, len(buf))
    return buf.value.decode("utf-8")
```

- pointer **and** length in, caller-owned buffer out
- returns the length it needed, like `snprintf`
- UTF-8: "Zoë" is 3 characters, 4 bytes

Rules

- The caller owns the memory: numpy allocates, Fortran reads or fills
- `flags="F_CONTIGUOUS"` for arrays Fortran modifies in place
- Keep ctypes callbacks alive while Fortran may call them
- Windows without conda: `fpm install --prefix . --link-flag -static`

3. Strings

Passing strings between Fortran and C

Two ideas of a string

	C	Fortran
storage	array of `char`	`character(len=n)`
length	up to the first NUL (`'\0'`)	carried with the variable
padding	none	trailing blanks

Every conversion bridges these rows.

Fortran to C: the interface

```
function c_length(s) bind(C) result(n)
    import :: c_char, c_size_t
    character(kind=c_char), intent(in) :: s(*)
    integer(c_size_t) :: n
end function c_length
```

`const char *` is a `character(kind=c_char)` array; a Fortran string can be passed to it.

Fortran to C: add the NUL

```
pure function f_to_c(f) result(c)
    character(len=*), intent(in) :: f
    character(kind=c_char, len=:), allocatable :: c

    c = f//c_null_char
end function f_to_c
```

```
len(word)                    = 10
c_length(f_to_c(word))       = 10
c_length(f_to_c(trim(word))) = 7
c_length(f_to_c('Zoë'))      = 4
```

Blanks count unless trimmed. Lengths are **bytes**.

C to Fortran: copy

```
function c_ptr_to_f(cptr) result(f)
    type(c_ptr), intent(in) :: cptr
    character(len=:), allocatable :: f

    character(kind=c_char), pointer :: chars(:)
    integer :: i, n

    if (.not. c_associated(cptr)) then
        f = ""
        return
    end if
    n = int(c_strlen(cptr))
    call c_f_pointer(cptr, chars, [n])
    allocate (character(len=n) :: f)
    do i = 1, n
        f(i:i) = chars(i)
    end do
end function c_ptr_to_f
```

A `c_ptr` has no length: ask `strlen`. The copy outlives the C memory.

C to Fortran: look in place

```
function c_to_f_pointer(cptr) result(fptr)
    type(c_ptr), intent(in) :: cptr
    character(kind=c_char, len=:), pointer :: fptr

    ! c_f_pointer cannot set a deferred length: go through a pointer of
    ! specified length, whose target the result then points at.
    fptr => with_length(cptr, int(c_strlen(cptr)))
end function c_to_f_pointer

function with_length(cptr, n) result(p)
    type(c_ptr), intent(in) :: cptr
    integer, intent(in) :: n
    character(kind=c_char, len=n), pointer :: p

    call c_f_pointer(cptr, p)
end function with_length
```

`c_f_pointer` cannot set a deferred length: go through a pointer of **specified** length.

C fills a Fortran buffer

```
needed = c_greet(f_to_c("Zoë"), small, size(small, kind=c_size_t))
print '(3a,i0,a)', "   small buffer                = '", c_to_f(small), "' (needed ", needed, " bytes)"
if (needed >= size(small)) then
    allocate (big(needed + 1))     ! + 1 for the NUL
    needed = c_greet(f_to_c("Zoë"), big, size(big, kind=c_size_t))
    print '(3a)', "   big buffer                  = '", c_to_f(big), "'"
end if
```

```
small buffer                = 'Hello, ' (needed 12 bytes)
big buffer                  = 'Hello, Zoë!'
```

Pass the size, respect it, report what was needed.

Arrays of strings

```
items(1) = "one"//c_null_char
items(2) = "two"//c_null_char
items(3) = "three"//c_null_char
do i = 1, size(items)
    item_ptrs(i) = c_loc(items(i))
end do
print '(a,i0)', "   c_total_length(3, items)     = ", c_total_length(3, item_ptrs)
```

`char **` is an array of `type(c_ptr)`, each the `c_loc` of a `target`.

A non-conforming idiom

```
allocate (character(len=c_strlen(cptr)) :: fptr)
call c_f_pointer(cptr, fptr)
```

- `c_f_pointer` forbids a deferred-length `fptr` (F2018 16.9.41)
- it also leaks the allocation
- it happened to work with ifort, and old code relies on it

Three compilers, three behaviours

```
fpm run --example strings_c2f_pitfall
```

- **gfortran 16**: still “works”, silently
- **ifx 2025.3**: keeps the length, points at garbage

```
len = 5, value = <0T▯▯>
```

- **flang 22**: refuses to compile it, the best outcome

```
error: FPTR= argument to C_F_POINTER() may not have a deferred type parameter
```

Fix: copy, use a specified-length pointer, or F2023 `c_f_strpointer` (ifx only so far).

Rules

- Append `c_null_char` for C, `trim` when blanks are padding
- `strlen` returns `size_t`: `integer(c_size_t)`, not `int`
- Copy by default; alias C memory only if you control its lifetime
- Buffers have a size: pass it, respect it, report truncation
- Lengths are bytes, not characters

Conclusions

- **One door**: `bind(C)` and C types. The contract is a list of C prototypes.
- **Plugins**: check a version first, keep ownership explicit, close the library yourself. Each library brings its own Fortran runtime.
- **Python**: a plain Fortran core, a thin `bind(C)` layer, and numpy checks of type, shape and memory order in the wrapper.
- **Strings**: C counts on a NUL, Fortran on a length. Copy, pass the lengths, let the caller own the buffer.
- **Descriptors** (`x(:)`, `character(*)`): one compiler only. `optional` is fine everywhere.
- **Compilers**: support varies. gfortran and ifx are the most robust, yet pitfalls exist, and mixing compilers is restricted to the C subset.