Fortran 2003: C Interoperability

Salvatore Filippone

School of Aerospace, Transport and Manufacturing,
salvatore.filippone@cranfield.ac.uk

IT4I, Ostrava, April 2016
The need for interoperability

“Woe be to him that reads but one book.”
George Herbert

The C programming language underlies most modern operating systems; this in turn has made it a lingua franca in which to express all kinds of interactions with the computing environment.
What needs to be done:

- Data types and their representations (on both sides of the street);
- Calling conventions (including function/subroutine names);
- Tricks, traps and pitfalls.

The interoperability features are defined in an intrinsic module

use iso_c_binding
Basic data types: integers

Simple types interoperate through prearranged KINDs. Predefined integer KIND constants:

<table>
<thead>
<tr>
<th>Fortran kind constant</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_int</td>
<td>int</td>
</tr>
<tr>
<td>c_short</td>
<td>short int</td>
</tr>
<tr>
<td>c_long</td>
<td>long int</td>
</tr>
<tr>
<td>c_long_long</td>
<td>long long int</td>
</tr>
<tr>
<td>c_signed_char</td>
<td>signed char</td>
</tr>
<tr>
<td>c_size_t</td>
<td>size_t</td>
</tr>
<tr>
<td>c_int8_t</td>
<td>int8_t</td>
</tr>
<tr>
<td>c_int16_t</td>
<td>int16_t</td>
</tr>
<tr>
<td>c_int32_t</td>
<td>int32_t</td>
</tr>
<tr>
<td>c_int64_t</td>
<td>int64_t</td>
</tr>
<tr>
<td>c_int_least8_t</td>
<td>int_least8_t</td>
</tr>
<tr>
<td>c_int_least16_t</td>
<td>int_least16_t</td>
</tr>
<tr>
<td>c_int_least32_t</td>
<td>int_least32_t</td>
</tr>
<tr>
<td>c_int_least64_t</td>
<td>int_least64_t</td>
</tr>
<tr>
<td>c_int_fast8_t</td>
<td>int_fast8_t</td>
</tr>
<tr>
<td>c_int_fast16_t</td>
<td>int_fast16_t</td>
</tr>
<tr>
<td>c_int_fast32_t</td>
<td>int_fast32_t</td>
</tr>
<tr>
<td>c_int_fast64_t</td>
<td>int_fast64_t</td>
</tr>
<tr>
<td>c_intptr_t</td>
<td>intptr_t</td>
</tr>
</tbody>
</table>
Basic data types: real, complex

Predefined KIND constants:

<table>
<thead>
<tr>
<th>Fortran kind constant</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_float</td>
<td>float</td>
</tr>
<tr>
<td>c_double</td>
<td>double</td>
</tr>
<tr>
<td>c_long_double</td>
<td>long double</td>
</tr>
<tr>
<td>c_float_complex</td>
<td>float _Complex</td>
</tr>
<tr>
<td>c_double_complex</td>
<td>double _Complex</td>
</tr>
<tr>
<td>c_long_double_complex</td>
<td>long double _Complex</td>
</tr>
<tr>
<td>c_bool</td>
<td>_Bool</td>
</tr>
<tr>
<td>c_char</td>
<td>char</td>
</tr>
</tbody>
</table>

Types being interoperable means a variable declared in one language can be passed and understood correctly to the other.
If you have C functions with a header file like this:

```c
int gcd(int m, int n);
void foo(float *v, int *k);
```

you can do the following:

```fortran
use iso_c_binding
integer m,n,k,l;
real :: v(10)

interface
  function gcd(m,n) result(res) bind(c)
    use iso_c_binding
    integer(c_int), value :: m,n
  end function gcd

  subroutine foo(v,k) bind(c)
    use iso_c_binding
    integer(c_int) :: k
    real(c_float) :: v(*)
  end function foo
end interface

l = gcd(m,n)
call foo(v,k)
```
Variables are interoperable if they are of interoperable type; that is, they are “understood” by both the Fortran and the C side.

For arrays:

*A Fortran array of an interoperable base type and of shape $(e_1, e_2, \ldots, e_r)$ is interoperable with a C array with the corresponding base type and shape with reversed subscript order*

As an example the declaration

```fortran
integer(c_int) :: fa(10,2:3), fb(10,2:3,4)
```

corresponds to

```c
int ca[2][10], cb[4][2][10];
```
Derived types

When it comes to interoperability, derived types are quite restricted. To be interoperable a derived type:

- Must have the bind attribute;
- Must not be a sequence type;
- Must not have type parameters;
- Must not have any type-bound procedures, nor have the extends attribute;
- Must not be a pointer, nor allocatable, nor a zero-sized array.

Under these rules we can define

```c
typedef struct {
    int   m, n;
    float r;
} myctype;
```

```c
use, intrinsic :: iso_c_binding
type, bind(c) :: myftype
    integer(c_int) :: m, n
    real(c_float) :: r
end type myftype
```
No C program can do without pointers.

Basics:
- A C pointer has a Fortran interoperable type:
  ```fortran
  type(c_ptr) :: cp=c_null_ptr
  type(c_funptr) :: cfp
  ```
- A Fortran pointer does *not* have a C interoperable type because C pointers are just addresses, Fortran pointers are much more complex (currently/recently debated in the standards committee).

How do you build such pointers?

```fortran
real, target :: v(10)
cp = c_loc(v)
cfp = c_funloc(my_interoperable_function)
```

A `c_ptr` is the equivalent of a `void *` and should be *cast* to be used.
Walking your pointers

We have already seen the value attribute; consider the difference between

```c
void foo( void *p);
use, intrinsic :: iso_c_binding
interface
    subroutine foo(p) bind(c)
        type(c_ptr), value :: p
    end subroutine foo
end interface
```

and

```c
void foo( void **p);
use, intrinsic :: iso_c_binding
interface
    subroutine foo(p) bind(c)
        type(c_ptr) :: p
    end subroutine foo
end interface
```

Confusing the two is a common blunder.
Walking your pointers

The full rules for \texttt{c_loc}; its argument must either:

- Be of interoperable type and type parameters, and be either:
  1. A variable with the \texttt{target} attribute;
  2. An allocated allocatable with \texttt{target} and not zero-sized array;
  3. An associated scalar pointer.

- Be a non-polymorphic scalar, have no length parameters and be either:
  1. A non allocatable, non pointer with \texttt{target};
  2. An allocated allocatable with \texttt{target};
  3. An associated pointer;

In particular, you can \textit{not} pass a polymorphic type back and forth (the dynamic type would get lost), but don’t despair just yet! That’s just another use for the STATE design pattern.
To call a C function from Fortran, you must:

1. Have an explicit interface;
2. Use the bind(c) attribute;
3. Arrays on the C side must be explicit or assumed size (possibly copy-in/copy-out);

An example:

```fortran
interface
    subroutine foo(v,k) bind(c,name='C_Foo')
        integer(c_int), value :: k
        real(c_float) :: v(*)
    end function foo
end interface
```

```fortran
void C_Foo(float *v, int k);
```

is (almost certainly) compatible with

```fortran
use iso_c_binding
integer m,n,k,l;
real :: v(10)
```

```fortran
call foo(v,k)
```
Calling Fortran from C

Suppose you have an array declared in C, and you want to use it in a Fortran subroutine; you can do the following:

On the C side

```c
void F_Foo(int n, void *v);
float *v;
int n;
n=1000;
v=(float *) malloc(n*sizeof(float));
init_array(n,(void *)v);
F_Foo(n,v);
```

and on the Fortran side:

```fortran
subroutine foo(n,v) bind(c,name='F_Foo')
  use iso_c_binding
  integer(c_int), value :: n
  type(c_ptr), value :: v
  real(c_float), pointer :: fv(:)
  call c_f_pointer(v,fv,(/n/))
end function foo
```

S. Filippone (SATM)
Can you make two Fortran subroutines communicate via a C caller? YES! Since `type(c_ptr)` is a `void *`, it can point to a non-interoperable type that C will treat as an opaque object.

```fortran
module foobar
  type my_private_type
    integer, allocatable :: v(:)
  end my_private_type
contains
  function foo() result(this) bind(c,name='F_foo')
    use, intrinsic :: iso_c_binding
    type(c_ptr) :: this
    type(my_private_type), allocatable :: item
    allocate(item)
    allocate(item%v(100))
    this = c_loc(item)
  end function foo
  subroutine bar(this) bind(c,name='F_bar')
    type(c_ptr),value :: this
    type(my_private_type), pointer :: item
    call c_f_pointer(this,item)
    .......
  end subroutine bar
end module
```