Lecture VI: Some more advanced features

Outline

- Interoperability with C/C++
- Reading environment variables
- Executing system commands
  - Most slides originate from a lecture by Sami Saarinen, CSC

Language interoperability

- Problem of language interoperability been present for long time
- Utilizing libraries or other pieces of written in Fortran from C programs or vice versa
  - Solvers, I/O schemas, database interfaces,...
- Realities of academic software collaborations...

Example: Calling a C library (CBLAS) from Fortran

```
program f_calls_c
use, intrinsic :: iso_c_binding
implicit none
integer(kind=c_int), parameter :: n = 100
real(kind=c_float) :: x(n), y(n), sdot
interface
  function cblas_example(n, x, incx, y, incy) bind(c, name='cblas_sdot')
     result(anumber)
    use, intrinsic :: iso_c_binding
    integer(kind=c_int), value :: n, incx, incy
    real(kind=c_float), intent(in) :: x(*), y(*)
    real(kind=c_float) :: anumber
  end function cblas_example
end interface
! init x & y
x(:) = 1.5  ; y(:) = 2.5
sdot = cblas_example(n, x, 1, y, 1) ! a direct call to 'cblas_sdot'
print *,'sdot=',sdot
end program f_calls_c
```

The bottom line is to use INTERFACE-blocks and **bind(c)**

Basic remarks

- A Fortran SUBROUTINE is mapped to a C-function with *void* result
  - A Fortran FUNCTION on the other hand maps to a C-function that *returns* a value
- The main program can be either in Fortran or C language
- Binding <label> in **bind(c, name=<label>)**
  - Is case sensitive when provided (e.g. **name='C_funcX'**)
  - If the "**name=**" attribute is omitted, it takes Fortran name converted to lowercase (no underscores appended)

The ISO_C_BINDING module

- Activated via

  ```
  use, intrinsic :: iso_c_binding
  ```
- Provides access to named constants that represent kind type parameters of data representations compatible with C types
- The module also contains
  - the derived type **`c_ptr`** corresponding to any C data pointer type
  - the derived type **`c_funptr`** corresponding to any C func. pointer type
- Also contains few helper routines, e.g.
  - **`c_loc, c_funloc, c_f_pointer, c_associated`**

Mapping between Fortran and C data types

- Using ISO C-bindings consistently, we can make sure our codes run correctly, despite "multi-lingual" nature
- The most common intrinsic data types mappings

Fortran declaration	C data type
INTEGER(c_short)	short int
INTEGER(c_int)	int
INTEGER(c_long_long)	long long int
REAL(c_float)	float
REAL(c_double)	double
CHARACTER(1,c_char)	char

Array data interoperability

- C-array indexing always starts from index zero (0)
  - Fortran default indexing starts from one (1)
- Multidimensional arrays in C "grow fastest" along the last dimension e.g. 2D-arrays are "row-major"
  - Fortran multidimensional arrays are opposite, e.g. 2D-arrays are "column-major"

```
! fortran array declarations compatible with c

real (c_double) :: z1(5) ! indexing starts at 1
real(c_float)   :: z2(4:6,17) ! 3 x 17 matrix
integer (c_int) :: ivec(-4 : 7) ! 12 elements
```

```
/* corresponding c-declarations */

double z1[5]; ! indexing between 0 and 4
float  z2[17][3] ; ! note index swap
int  ivec[12] ; ! indexing between 0 and 11
```

Accessing C data structures from Fortran

- In many cases it is possible to describe Fortran derived data types in terms of C data structures (and vice versa)
- You need ISO C binding module again plus Fortran derived type must have the bind(c) attribute
  - However, use of **sequence** keyword is forbidden
- Each individual Fortran data type component must also be of an interoperable type
- Fortran components cannot be allocatables or pointers
- C types cannot be unions nor structures with bit-fields

Accessing C data structures from Fortran

```
/* C data structure */

typedef struct {
    int count ;
    double d [ 100 ] ;
} C_type;
```

```
MODULE my_typedef
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
TYPE, bind(c) :: C_type_as_seen_by_Fortran
    integer(kind = c_int) :: count      ! Can be any var. name
    real(kind = c_double) :: d ( 100 ) ! Can be any var. name
END TYPE C_type_as_seen_by_Fortran
END MODULE my_typedef
```

- Note that variable ordering, data types and fixed array sizes must be identical
- Typical usage comes through function calls, e.g. Fortran extracting information from a C-routine

```
/* C-function example */

void C_func(C_type *p) {
    p->count = 1;
    p->d[0] = 1.23;
}
```

```
USE my_typedef
INTERFACE
    SUBROUTINE TESTF(P) bind(c, name='C_func')
    USE my_typedef
    TYPE(C_type_as_seen_by_Fortran) :: P
    END SUBROUTINE TESTF
END INTERFACE
TYPE(C_type_as_seen_by_Fortran) :: X
CALL TESTF(X)
PRINT *,X % count, X % d(1)  ! NB: Fortran indices 1...100
```

Accessing dynamic data components found in C

```c
/* C data structure */

typedef struct {
    int count ;
    double *d ; // dynamic
} C_type;
```

```fortran
MODULE my_typedef
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
TYPE, bind(c) :: C_dynamic
    integer(kind = c_int) :: count
    type(kind = c_ptr) :: d          ! Maps to “double *d”
END TYPE C_dynamic
END MODULE my_typedef
```

- Note : use of TYPE(c_ptr) and utility routine C_F_pointer ( ) are crucial ! !

```c
/* C-function example */
#include <stdlib.h> // for malloc
void C_func(C_type *p, int n) {
    p->count = n;
    p->d = malloc(n * sizeof(*p->d)) ;
    if (n > 0) p->d[0] = 1.23;
}
```

```fortran
USE my_typedef
INTERFACE
SUBROUTINE TESTF(P,N) bind(c, name=‘C_func’)
     USE my_typedef
     TYPE(C_dynamic) :: P
     INTEGER(c_int), value :: N
     END SUBROUTINE TESTF
END INTERFACE
TYPE(C_dynamic) :: X
INTEGER(c_int), PARAMETER :: N = 10
REAL(c_double), POINTER :: xd(:)
CALL TESTF(X,N)
CALL C_F_pointer (x % d, xd, (/ N /) )   ! Mapping !!
PRINT *,X % count, xd(1)
```

Note: cannot map Fortran allocatable components

- Please note that any attempt to use non-fixed size components and relate it to Fortran ALLOCATABLE is doomed because C and Fortran pointers are entirely different concepts
- Thus the following attempt *does not work* – period

```
/* C data structure */

typedef struct {
    int count ;
    double *d ; // dynamic
} C_type;
```

```
module my_typedef
use, intrinsic :: iso_c_binding
implicit none
type, bind(c) :: c_dynamic
    integer(kind = c_int) :: count
    real(c_double), allocatable :: d ( : )
end type c_dynamic
end module my_typedef
```

Exchanging global data

- Global data may be defined in Fortran in terms of data in the Fortran modules, or in COMMON blocks
- In C language they are declared once outside functions (often in **`main`**) and referenced via **`extern`**'s elsewhere
  - These can be accessed in Fortran by using **`bind(c)`** label
- Fortran module data is generally impossible to access from C as Fortran module names are compiler and linker depended
- Consistently defined variables in Fortran COMMON blocks can also be accessed in C side

Some compatible global data mappings

```
/* Global C data */

int number ;

float Array[8] ;

double slice [3] [7];

struct coord {
  float x, y, z;
};

struct coord xyz ;
```

$\longleftrightarrow$

```
! fortran global data must sit in a module

module something
use , intrinsic :: iso_c_binding
implicit none

integer(c_int), bind(c) :: number

real(c_float) :: my_array(8)
bind(c, name='array') :: my_array

real(c_double), bind(c) :: slice(7,3) ! index
    swap

real(c_float) :: x, y, z
common /xyz/ x, y, z
bind(c) :: /xyz/          ! note /…/ syntax

end module something
```

Handling (binary) I/O

- Restricting us to binary (unformatted) I/O only
  - Formatted text files are usually not a concern
- Fortran unformatted, non-direct access files by default contain record delimiters (4 or 8 bytes long)
  - Note: STREAM I/O and direct access files do not have them
- Files written from C-language don't have record delims
- Could be a real headache
- Writing & reading files with STREAM I/O in Fortran usually solves most of the problems

Handling binary I/O

```
use, intrinsic :: iso_c_binding
:
integer(kind = c_int) :: array(100)

open (10, file='file.bin',form='unformatted',
     access='stream',status='unknown')

write (10) array(1:100) ! write 400 bytes
write (10) array(1:50)  ! write 200 bytes

close (10)
```

File 'file.bin'– it's just data:

```
... ARRAY(1:100) ... ...
... ARRAY(1:50) ...
```

```
/* The corresponding C-reader is
    trivial */
int array[100];

FILE *fp = fopen ("file.bin", "r");

fread (array, sizeof(*array), 100, fp);
fread (array, sizeof(*array), 50, fp);

fclose (fp)
```

- The key: **ACCESS = 'STREAM'**
- No artificial record delimiters

Interoperability: Conclusions

- Fortran standard now officially supports mechanisms to call source codes or libraries written in C language, as well as define Fortran routines to become callable from C
  - There seems to be a number of pitfalls
  - Dynamically allocated entities a culprit
  - Use interoperability with care and avoid complicated structures and calling sequences
- When new STREAM I/O access is used in Fortran, binary files at least become interoperable with C-language

Environment variables

- Besides command line arguments, *environment variables* are a common way to modify program behaviour
- Fortran 2003 has a standardized method for accessing values of environment variables
- In Fortran 77/90/95 accessing **`getenv`** from C standard library requires passing character strings from Fortran to C and back

Environment variables

- Access a value of an environment variable

```
call get_environment_variable(name,value[,length]
                              [,status][,trim_name])
```

  - **`name`** is of type character string and contains the name of the requested variable
  - **`value`** is of type character string and contains the value of the requested variable. If the the actual variable value is too short or long it is padded with blanks or truncated. If the variable has no value or does not exist, **`value`** is set to blanks
  - **`length`** is of type integer and contains the length of the requested variable on return if the variable exists and has a value and zero otherwise (optional)
  - **`status`** is type integer. If requested variable does not exist status is 1. If **`value`** was too short status is -1 and zero otherwise. For other return codes, see docs (optional)
  - **`trim_name`** is of type logical and sets if trailing blanks are allowed in variable names or not (optional)

Environment variables: example

```
program environment
implicit none
character(len=256) :: enval
integer :: len,stat
call get_environment_variable('HOSTNAME',enval,len,stat)
if (stat == 0) write (*,'(A,A)') 'Host=', enval(1:len)
call get_environment_variable('USER',enval,len,stat)
if (stat == 0) write (*,'(A,A)') 'User=', enval(1:len)
end program environment
```

Executing commands

- Invoking *external (other) programs* and *system commands* from within a program is sometimes useful
  - No source nor library API available for a useful program
  - Pre/post processing scripts
  - MPMD
- Fortran 2008 has a standardized method for invoking an external command
- In Fortran 77/90/95 accessing **`system`** from C standard library requires passing character strings from Fortran to C and back
  - With Fortran 2003, ISO-C bindings can be used

Executing commands

- Execute a command line

```
call execute_command_line(command[,wait][,exitstat]
                                [,cmdstat][,cmdmsg])
```

  - **`command`** is a character string containing the command to be invoked
  - **`wait`** is logical value indicating if command termination is to be waited (**`.true.`**, the default) or if the command is to be executed asynchronously (**`.false.`**) (optional)
  - **`exitstat`** is an integer value containing the return value of the command if **`wait=.true.`** (optional)
  - **`cmdstat`** is an integer value. It is assigned a value of zero if **`command`** executed successfully. For other return codes, see docs (optional)
  - **`cmdmsg`** is a character string containing explanatory message for positive values of **`cmdstat`** (optional)

Executing commands: example

```
program execcommand
implicit none
integer :: estat, cstat
call execute_command_line('ls -al', .TRUE., estat, cstat)
if (estat==0) write (*,'(A)') 'Command completed successfully’
end program execcommand
```

Programming assignment

The tarball snapshot.tar.gz provides an interface for calling the utility libpng (which, e.g., generates png images from data files) from Fortran programs. The program is courtesy of Elias Toivanen (UH).

Some crucial parts have been removed - your task is to provide the interfaces for calling the functions in matrix_snapshot_interop.c from Fortran such that the program works again and print nice .pbm images from matrix data.