Standardized Mixed Language Programming for Fortran and C

Bo Einarsson, Richard J. Hanson,
Tim Hopkins

1 September 2009

Contents

Abstract

Programmers have long practiced the matter of mixed language procedure calls. This is particularly true for the programming languages C and Fortran. The use of the alternate language often results in efficient running time or the effective use of human or other resources.

Prior to the Fortran 2003 standard there was silence about how the two languages interoperated. Before this release there existed a set of differing *ad hoc* methods for making the inter-language calls. These typically depended on the Fortran and C compilers. The newer Fortran standard provides an intrinsic module, `iso_c_binding`, that permits the languages to interoperate. There remain restrictions regarding interoperable data types.

This paper illustrates several programs that contain core exercises likely to be encountered by programmers. The source code is available from the NSC web site. Included is an illustration of a "trap" based on use of the *ad hoc* methods: A call from a C to a Fortran 2003 routine that passes a character in C to a character variable in Fortran results in a run-time error.

1. Introduction

At a conference in Kyoto 1995 a lecture about the use of routines in C from Fortran, and the opposite, was presented (Einarsson, 1995). At that time, it was then necessary to describe this differently for each different platform.

Fortran 2003 has standardized the mixing of Fortran and C with the concept of "Interoperability with C." Many Fortran compilers already implement parts of the 2003 standard, see (Chivers and Sleightholme, 2009). We look at a number of different aspects of mixed language programming and present examples to demonstrate the facilities that are available in the new standard.

We have checked all these examples using

1. Intel Fortran compiler 10.1

2. IBM XL Fortran Enterprise Edition for AIX, V11.1 (5724-S72) Version 11.01.0000.0001D, (September 19, 2007)

3. Sun Fortran 95 8.3 (July 18, 2007)

4. NAGWare Fortran 95 compiler Release 5.2 (668)

5. g95 version 0.92 (March 14, 2009)

6. gfortran version 4.4.0 (February 19, 2009)

7. The Portland Group, pgf90 8.0-5 and pgf95 9.0-2, 64-bit target on x86-64 Linux

There is an old recommendation for mixed language programming (or inter-language communication) that only one of the languages should be used for input and output. We violate this recommendation in the last example of Section 4, where output is generated both by Fortran and C. The order of the output may differ on different platforms due to the fact that both Fortran and C may buffer output for efficiency reasons. This is especially noticeable when output is redirected to a file. This problem can often be ameliorated by judicious use of the flush statement in both languages.

The 2003 standard allows data to be exchanged between Fortran and C provided that it is represented and interpreted in the same way in each language. The tables given in (Metcalf et al., 2004, Table 14.1) and (Adams et al., 2008, Table 15-1) detail how to choose the correct Fortran kind values to ensure compatibility between the two languages.

We restrict the use of these KIND values to the small wrapper routines that we write to implement cross calling.

The BIND attribute and the iso_c_binding intrinsic module provide the necessary facilities to write standard conforming code that will allow Fortran and C components to interoperate. It may also be the case that a Fortran compiler requires a specific C compiler to be used in order for interoperability to take place.

Note that there may be traps for the unwary when using a Fortran 2003 compatible compiler to compile and link mixed Fortran and C code that was written to make use of the old platform-dependent mechanisms. We have encountered examples where the combination of a new compiler and *ad hoc* methods cause unexpected run-time errors. Typically these are caused by problems relating to the length of string arguments.

A simplified form of an example that caused such as error is given by these code fragments:

```
SUBROUTINE fortran(char)
 CHARACTER(LEN=1) char
    ...
```

```
extern void fortran_(char* a)
...
fortran_("L");// Here fortran_ is
              // the mangled external
              // name provided by the
              // Fortran compiler,
              // but without using
              // interoperability
              // standards.
```

The calling C routine has no direct language support for specifying the length of the Fortran character argument "L". The Fortran run-time system, if it is "picky," can certainly check that the length of this argument has a value *at least one*. By chance the value in memory might be positive, but otherwise arbitrary. Or the Fortran system can assume that the length of the character is exactly one and make no check.

But if by chance the arbitrary value is non-positive, then the "picky" Fortran system can rightly issue an error. The NAGWare Fortran 95 compiler gives an instance where exactly this error occurs.

With the Intel and Portland compilers when the main program is in C, it is recommend to link with C instead of with Fortran, if not the system complains about double main programs. The linking with C has to include some Fortran utilities.

Our first example checks to see if the basic data types are compatible between the two languages. With the exception of the Portland compiler*, running the code given on the next two pages gives the results reported on the following page.

*The Portland 8.0 and 9.0 compilers give an error message that a `KIND` value is negative, indicating a not implemented data type.

```
PROGRAM CandF
! Program to test cooperation
! between C and Fortran
USE, INTRINSIC :: iso_c_binding

  CALL check(c_int, KIND(1), &
             'integer/int')
  CALL check(c_float, KIND(1.0e0), &
             'real/float')
  CALL check(c_double, KIND(1.0d0), &
             'double precision/double')
  CALL check(c_bool, KIND(.TRUE.), &
             'logical/boolean')
  CALL check(c_char, KIND('A'), &
             'character/char')
  CALL check(c_float_complex, &
            KIND((1.0e0, 1.0e0)), &
             'complex/float_complex')

END Program CandF
```

```
SUBROUTINE check(ckind,fkind,vartype)
INTEGER, INTENT(IN) :: ckind, fkind
CHARACTER(LEN=*), INTENT(IN)::vartype

IF (ckind == fkind) THEN
 WRITE(*,'(''Default Fortran &
  &and C variables of type '',&
  &a, ''are interoperable'')')&
  & vartype
ELSE IF (ckind < 0) THEN
 WRITE(*, '(''A compatible Fortran,&
  & kind value is not , &
  & available for type '',a)') vartype
ELSE
 WRITE(*,'(''Default Fortran and C &
  &variables of type '', a, &
  &'' are NOT interoperable'')') &
  & vartype
END IF

END SUBROUTINE check
```

Default Fortran and C variables of type
integer/int are interoperable
Default Fortran and C variables of type
real/float are interoperable
Default Fortran and C variables of type
double precision/double are interoperable
Default Fortran and C variables of type
logical/boolean are NOT interoperable
Default Fortran and C variables of type
character/char are interoperable
Default Fortran and C variables of type
complex/float_complex are interoperable

The value returned for `c_bool` indicates that a compatible `LOGICAL(KIND=c_bool)` is provided by the compiler but this must be explicitly declared as it differs from the KIND value associated with the default `LOGICAL`. In this instance we can arrange to convert between the Fortran and C types using intrinsic overloaded assignment in the communication (or wrapper) routine. For example

```
...
USE, INTRINSIC :: iso_c_binding
! Declare types for logical variables
   LOGICAL           :: fortran_logical
   LOGICAL(c_bool) :: c_boolean
! Transfer value of Fortran LOGICAL
! to C c_bool type
   c_boolean = fortran_logical
! Transfer in the opposite direction
   fortran_logical = c_boolean
...
```

With version 8.0 (or 9.0-2) of the Portland compiler a system dependent solution has to be used. For example a wrapper C routine can convert a `c_bool` or _Bool variables to `c_int` variables, with {0,1} values, that are communicated to Fortran.

2. Use of a Fortran subprogram from C

We begin by showing what we need to do to call a simple Fortran subroutine and function from C. This illustrates how a wrapper routine can be constructed to allow calls to be made from C even when the original source code to the Fortran procedures is not available.

We start by defining the two simple Fortran subprograms (sam.f90) and constructing a Fortran driver program (f2sam.f90) to call them.

```
      SUBROUTINE sam(f, b, s)
      EXTERNAL f
      INTEGER f
      CHARACTER(LEN=7), &
          INTENT(OUT) :: s
      INTEGER, INTENT(OUT) :: b
      REAL :: x
      x = 1.3
      s = 'Bo G E '
      b = f(x)
      END SUBROUTINE sam

      INTEGER FUNCTION f(x)
      REAL, INTENT(IN) :: x
      f=3*x**3
      RETURN
      END FUNCTION f

!   Sample main program f2sam.f90 used
!   to check correct operation of the
!   subroutine sam and the function f
      PROGRAM fdriver
      EXTERNAI f
      INTEGER f
      CHARACTER*7 s
      INTEGER b(3)
      CALL sam(f, b(2), s)
      WRITE(6,'(i5,i5,10x,a7)') &
            b(2), f(REAL(b(2))), s
      END PROGRAM fdriver
```

We now look at what needs to be done to make the procedures `sam` and `f` callable from C without the need for any changes to the original code. We achieve this in (c_sam.f90) by constructing wrapper or communication routines in Fortran which call the original procedures.

First the function:

```
FUNCTION c_f(x) RESULT (f_res) &
          BIND(C,NAME='c_f')
USE, INTRINSIC :: iso_c_binding, &
          ONLY : c_int, c_float

INTEGER (c_int)    :: f_res
REAL(c_float)      :: x
INTEGER, EXTERNAL :: f
  f_res=f(x)
END FUNCTION c_f
```

And then the subroutine:

```
SUBROUTINE c_sam(c_f, b, s)  &
    BIND(C, NAME='c_sam')
USE, INTRINSIC :: iso_c_binding, &
     ONLY : c_char, c_int, c_null_char
INTERFACE
 FUNCTION c_f(x) RESULT(f_res) BIND(C)
 USE, INTRINSIC :: iso_c_binding, &
                   ONLY : c_int, c_float
 INTEGER (c_int)    :: f_res
 REAL (c_float)     :: x
 END FUNCTION c_f
END  INTERFACE

! C requires that the string be one
! character  longer than its Fortran
! equivalent as it needs to be null
! terminated.
  CHARACTER(KIND=c_char) :: s(8)
  CHARACTER(len=7)       :: t
  INTEGER (c_int)        :: b
  INTEGER                :: i

  CALL sam(c_f,b,t)
  DO i=1,7
     s(i)=t(i:i)
  END DO
  s(8)=c_null_char
! Terminate C string with
! null character
END SUBROUTINE c_sam
```

We draw attention here to two important requirements in the interoperability of character parameters. Only a scalar or an array of `CHARACTER(LEN=1)` is interoperable although the standard does allow an actual argument of length greater than one to be used when the dummy argument is an array of `CHARACTER(LEN=1)`. Second, C expects that strings will be terminated with the null character; failure to do this may result in fatal errors if C attempts to process the string with, for example, `strcpy`. This means that a C string will be one character longer than its equivalent Fortran string.

Finally, in (c2sam.c) we give the C program that calls the original routines `sam` and `f` via the wrapper routines `c_sam` and `c_f` respectfully. Note that the function prototypes of any Fortran routines called by the C program must be included.

```c
#include <stdio.h>

/* Function prototypes to wrapper */
/* routines to be called from C */

 int c_f(float *);
 int c_sam(int (*c_f)(float *),
           int *, char []);

int main()
{
 char s[8];
/* String is one entry longer */
/* for null ending */
 int b[3];
 float x;

/* Fortran wrapper routine */
/* c_sam calls sam */
   c_sam(c_f, &b[1], s);
   x = b[1];
   printf("%5d%5d%7s\n", b[1],
          c_f(&x), s);
   return 0;
}
```

We may compile and run these on a Unix/Linux system using a set of commands of the form

```
cc -c c2sam.c
f95 c_sam.f90 sam.f90 c2sam.o
a.out
```

to give the result

```
6  648          Bo G E
```

as before.

3. Use from Fortran of a matrix in C

In this case we wish to initialize elements of an array via a function written in C and use these values in a Fortran program. Because the Fortran standard mandates that array elements are stored in column major order and C requires their storage in row major order, we need to reverse the order of the indices to ensure that both languages use the same ordering of elements within memory.

In addition, we need to remember that in C an array declared `a[3][2]` defines elements `a[0][0]` ... `a[2][1]` and we must adjust our indexing accordingly. The C routine is given in (mlp4.c) while the Fortran main program is provided in (mlp3.f90).

```
void p(float a[3][2], int *i, int *j)
{
  a[*j-1][*i-1] = *i + *j/10.0;
/* Indices reduced by 1 */
}
```

```
PROGRAM mlp3

INTERFACE
  SUBROUTINE p(a,i,j) BIND(C,NAME='p')
    USE, INTRINSIC :: iso_c_binding, &
          ONLY : c_float, c_int
      REAL (c_float)     :: a(2,3)
      INTEGER (c_int)    :: i, j
  END SUBROUTINE p
END INTERFACE

! Declare a to be dimensioned 2,3
! to match C declaration of 3,2
      REAL ::  a(2,3)
      CALL p(a,1,3)
      WRITE (6,'(1x,dc,f9.1)') a(1,3)
! dc not accepted by Portland
END PROGRAM mlp3
```

4. Use of Fortran COMMON data in a C program

As in the first example we wish to use a Fortran routine from both Fortran and C, therefore we once again need a communication routine. The COMMON block is assigned its values in the routine `init_name` (mlp2a.f90) which together with the Fortran driver in (mlp0.f90) give the result 786 3.2.

```
      SUBROUTINE init_name()
      COMMON /name/ i, r

      i = 786
      r = 3.2

      RETURN
      END SUBROUTINE init_name
```

```
      PROGRAM mlp0
      INTEGER i
      REAL r
      COMMON /name/ i, r

      CALL init_name
      WRITE(*,'(i4, f10.3)') i, r

      END PROGRAM mlp0
```

If we **only** wish to use the routine `init_name` from C it is quite simple, we just modify `init_name` to the routine `init_read` as shown in (mlp2b.f90) and use the C program, given in (mlp1.c) to obtain the same result.

```
MODULE com_init_read
USE, INTRINSIC :: iso_c_binding, &
                  ONLY : c_int, c_float
INTEGER(c_int) :: i
REAL(c_float)  :: r
COMMON /com/ i, r
BIND(c) :: /com/
END MODULE com_init_read

SUBROUTINE init_read() &
       BIND(C,NAME='init_read')
USE com_init_read

      i = 786
      r = 3.2

END SUBROUTINE init_read
```

```c
#include <stdio.h>

/* Use of COMMON Block from Fortran */

struct  {
    int i; float r;
} com;

void init_read();

int main()
{
 init_read();
 printf("%4d%10.3f\n",com.i,com.r);
 return 0;
}
```

However the standard requires that if the `BIND` attribute is associated with one instance of a common block, it must be associated with all instances of that common block throughout the code. Thus, if we do not have access to the complete source code (e.g., we only have a pre-compiled library) providing the `BIND` attribute to a single definition of a common block that is used elsewhere will result in a non-conforming program. Even if the source code is available the changes required can be both numerous and error prone.

We therefore consider a second approach which uses a pointer and makes use of the fact that data stored in a common block must occupy a contiguous block of memory, see (Adams et al., 2008, section 5.14.2). We note here that this approach works best when all occurrences of the same common block have the same data layout.

The C routine is given in (mlp7.c) with the Fortran driver in (mlp6.f90). Here we also let the C routine change the values in the COMMON block.

```c
#include <stdio.h>
/* Use of COMMON Block from Fortran */

/* This is the layout of the common */
/* block in Fortran: */
typedef struct {
  int i; float r;
} comv;

/* Function prototypes for Fortran */
/* wrapper routines */

void *init_name_c();

void mc()
{
  comv *p;

/* Get address of common block */
/* from Fortran code. */
  init_name_c(&p);
  printf("C␣gets␣/com/:␣%4d%6.2f\n",
  p->i,p->r);
/* Change values in common block. */
  p->i = 457;
  p->r = 17.5;
  printf("C␣gives␣/com/:␣%4d%6.2f\n",
  p->i,p->r);
}
```

```
PROGRAM main_fortran
USE, INTRINSIC :: iso_c_binding, &
           ONLY : c_int, c_float

 INTEGER(c_int) :: i
 REAL(c_float)  :: r
 COMMON /com/ i, r

 INTERFACE
    SUBROUTINE init_name
    END SUBROUTINE init_name
    SUBROUTINE mc() BIND(C,NAME='mc')
    END SUBROUTINE
 END INTERFACE

 CALL init_name
! load COMMON block in Fortran
 CALL mc
! Call C code that gets pointer
! to COMMON
 WRITE(*,'(a,i4,f6.2)') &
 &'Fortran has the block /com/: '&
 &, i, r

END PROGRAM main_fortran
```

```
SUBROUTINE init_name_c(comp) &
      BIND(C,NAME='init_name_c')
USE, INTRINSIC :: iso_c_binding, &
  ONLY : c_ptr, c_int, c_float, c_loc
! The C function mc calls
! init_name_c() to get the location
! of the COMMON block start.
    TYPE(c_ptr) :: comp

    INTEGER(c_int), TARGET :: i
    REAL(c_float)          :: r
    COMMON /com/ i, r

! Return a pointer to the first
! location of the COMMON block.
    comp = c_loc(i)
    WRITE(*,'(a,i4,f6.2)')&
    &'Fortran has common /com/: ',&
    & i, r

END SUBROUTINE init_name_c
```

```
SUBROUTINE init_name
! This is the original Fortran
! function assigning values to
! the COMMON BLOCK

     INTEGER :: i
     REAL    :: r
     COMMON /com/ i, r

     i = 786
     r = 3.2

END SUBROUTINE init_name
```

This program may be compiled using

```
cc  -c mlp7.c
f95  mlp6.f90 mlp7.o
a.out
```

which runs to give

```
Fortran has common block /com/:  786  3.20
C gets Common block /com/:  786  3.20
C gives Common block /com/:  457 17.50
Fortran has the block /com/:  457 17.50
```

5. Use of a C routine from Fortran

As in the example above we wish to use a routine in C from a main program in Fortran, but in this case the C routine is a simple function.

As a simple example we write a function in C (mean.c) to evaluate the mean value of its two arguments. This may be called via the main program in (c_mean.c).

```
extern float mean (float x, float y);

/* Function to evaluate the mean value of

float mean (float x, float y)
{
    return ((x + y) / 2);
}
```

```c
#include <stdio.h>
/* Function to evaluate */
/* the mean value */
/* of two arguments */

float mean (float x, float y);

int main()
{
   float a, b, c;
   a = (float) 1.0;
   b = (float) 3.0;
   c = mean(a,b);
   printf("␣Mean␣value␣is␣␣%f␣\n␣", c);
   return 0;
}
```

When a procedure is called in Fortran, by default, all arguments are passed by reference; i.e., the address of the argument is passed through to the routine. In C it is possible to pass arguments either by value or by reference; the cases are distinguished by passing a variable or a pointer to a variable of the relevant type respectively.

To allow Fortran to interoperate successfully with C it is necessary to be able to distinguish between these two types of argument. The `VALUE` attribute, introduced in Fortran 2003, enables procedure arguments to be declared as called by value. This facility is available for defining the way in which parameters are passed among Fortran procedures as well as for interoperability with C.

In the function `mean` the two parameters, `x` and `y`, are both passed by value and hence, in the Fortran interface, these variables need to be declared with the value attribute. Failure to do this will result in the values being treated as addresses and either incorrect results or a fatal run-time error will result.

The complete program is provided in (mlp8.f90).

```
PROGRAM mlp8
USE, INTRINSIC :: iso_c_binding

INTERFACE
  REAL(c_float) FUNCTION mean(a,b) &
                BIND(C,NAME='mean')
  USE, INTRINSIC :: iso_c_binding
    REAL (c_float), VALUE :: a, b
  END FUNCTION mean
END INTERFACE

  REAL :: a, b, c

  a = 1.0
  b = 3.0
  c = mean(a,b)
  WRITE (*,*) ' Mean from C is ', c

END PROGRAM mlp8
```

6. GPU implementation

This example is compelling because the topic of using an attached Graphics Processing Unit (GPU) is now under intense discussion in the High Performance Computing (HPC) community. Some attractive benchmarking results (Barrachina et al., 2008) have been obtained from using an NVIDIA chip with BLAS (Dongarra et al., 1990). These routines are the basics of the LAPACK (Anderson et al., 1999) package.

We use a version of the matrix multiplication routine SGEMM that uses a GPU implementation of a routine available in the NVIDIA CUBLAS library (CUDA, 2008).

CUDA is a general purpose parallel computing architecture introduced by NVIDIA that enables the GPU to solve complex computational problems. It includes the CUDA Instruction Set Architecture (ISA) and the parallel compute engine in the GPU. To program to the CUDA architecture, developers can use C, which can then be run with great performance on a CUDA enabled processor. Other languages will be supported in the future, including Fortran and C++.

This example is noteworthy for three reasons:

1. The use of this level 3 BLAS code is well-known to be one of the most important for numerical linear algebra. At first glance it appears that one must replace calls to `SGEMM` with calls to `cublasSgemm`, the NVIDIA version. This can be avoided and is important when dealing with a pre-compiled application that uses `SGEMM`.

2. The NVIDIA documentation (CUDA, 2008) makes a statement that there is no standard interface for calling C functions from Fortran. The Fortran 2003 C interoperability facilities provide a solution to this issue.

3. Using a replacement code for `SGEMM` that turns around and calls the NVIDIA code is easy to write in a standard way. As we have preserved the BLAS name using this approach we need to ensure that the correct version of `SGEMM` is linked into the final executable rather than a 'standard' version that may be included in one of the standard pre-compiled libraries our application uses. Placing the required code ahead of any such libraries in the link command is usually enough to achieve this.

The example here uses the `VALUE` attribute for some scalar arguments and also transfers Fortran `CHARACTER` data to C char data which we note are single characters in the C specification. We include below a version of the `SGEMM` routine in (sgemm.f90). On the web site we also have a small driver program `sgemm_driver.f90` (also with explanation of all the arguments) and a C dummy routine `c_sgemm.c`. Our intent is to emulate the interface to the C NVIDIA code by using an identical specification for the arguments. NVIDIA wrote the C code to have column-oriented storage!

```
SUBROUTINE SGEMM(TRANSA,TRANSB,M,N, &
   &K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
USE ISO_C_BINDING
IMPLICIT NONE

! .. Scalar Arguments ..
  REAL ALPHA,BETA
  INTEGER K,LDA,LDB,LDC,M,N
  CHARACTER(LEN=*) TRANSA,TRANSB
  CHARACTER(KIND=c_char) CTA, CTB
! .. Array Arguments ..
  REAL A(LDA,*),B(LDB,*),C(LDC,*)
! Define the INTERFACE to the NVIDIA
! C code cublasSgemm.
! This version of SGEMM is used in a
! user application that calls LAPACK
! single precision routines, or makes
! other uses of that code.
  INTERFACE
! This is what the NVIDIA code
! expects for its inputs:
! void cublasSgemm (char transa,
! char transb, int m, int n, int k,
! float alpha, const float *A,
! int lda, const float *B, int ldb,
! float beta, float *C, int ldc)
  subroutine c_sgemm(transa, transb, &
  & m, n, k, &
  & alpha, A, lda, B, ldb, beta, c, &
  & ldc) bind(C,name='cublasSgemm')
```

```
  USE, INTRINSIC :: iso_c_binding, &
    ONLY : c_int, c_float, c_char

  character(KIND=c_char),value :: &
            transa, transb
  integer(c_int),value :: m,n,k, &
                          lda,ldb,ldc
  real(c_float),value :: alpha,beta
  real(c_float) :: A(lda,*), &
          B(ldb,*),C(ldc,*)
  end subroutine c_sgemm
  END INTERFACE

! The calculation, excepting
! initialization and finalization,
! is done with the NVIDIA C routine
! 'cublasSgemm.'
! A local name c_sgemm is used
! in Fortran.
! The name c_sgemm could be
! replaced by NVIDIA's name
! if one chose.
      cta = transa(1:1)
      ctb = transb(1:1)
      call c_sgemm(cta, ctb, &
       m, n, k, alpha, A, lda, &
       B, ldb, beta, c, ldc)
      return
!     ..
END SUBROUTINE SGEMM
```

In practice, to ensure maximum efficiency, the GPU would only be used when the problem size exceeded some break-even point; smaller problems would be dealt with directly on the master CPU, for example, using either the Intel MKL (Intel, 2008) or the vanilla code available from netlib (`http://www.netlib.org/blas/`). The break-even point is dependent on the GPU device being used and benchmarking would be needed to determine a suitable value.

The character arguments to `sgemm` are declared to be `CHARACTER(LEN=1)` in the definition of the routine in (Dongarra et al., 1990). It appears to be almost standard practice to use longer, more descriptive, constant, actual arguments; for example, `'TRANSPOSE'` rather than `'T'`. However, the Fortran standard states that only the first character of such arguments is used and thus the arguments to `sgemm` can be passed directly to `c_sgemm` provided that the default character type is interoperable. However, this does not work with all compilers, hence the use of the extra variables `cta` and `ctb`.

7. Enumerated Types

Fortran 2003 introduced a new enumeration definition mainly to allow interoperability with enumeration constants in C although it may be used freely within Fortran code. The standard guarantees that constants declared as type enumerator will correspond to the same integer type used by C, i.e., **int**.

An example of an enumeration in Fortran is shown below and is very similar to the **enum** definition in C.

```
MODULE enumdefs
     ENUM, BIND(C)
     ENUMERATOR :: jan=1, feb, mar, &
                   apr, may, jun, &
                   jul, aug, sep, &
                   oct, nov, dec
     END ENUM
END MODULE enumdefs
```

An enumerator constant may either be initialized explicitly (as jan above) or implicitly when it takes a value one greater that the previous enumerator constant in the list. If the first enumerator constant in the list is not explicitly initialized it is set to zero.

The Fortran standard provides no means of directly determining the kind selected for the integer used to store an enumerator value. Variables that may be assigned enumerator constants therefore need to be declared as, for example,

```
INTEGER(KIND(jan)) :: month
```

The simple Fortran subroutine and driver program given in (f_print_days.f90) and (f_main.f90) illustrate how the enumerator constants are used completely within Fortran.

```
SUBROUTINE print_days(month) &
     BIND(C, NAME='printDays')
USE enumdefs
INTEGER(KIND(jan)), VALUE :: month

SELECT CASE (month)
CASE(jan,mar,may,jul,aug,oct,dec)
 write(*,'(i2,''has 31 days'')') &
       month

CASE(apr,jun,sep,nov)
 write(*,'(i2,''has 30 days'')') &
       month

CASE(feb)
 write(*,'(i2,''has 28/29 days'')') &
       month

CASE DEFAULT
 write(*,'(i2,''does not exist!!'')')&
       month

END SELECT

END SUBROUTINE print_days
```

```
PROGRAM f_month
USE enumdefs, ONLY: jul, feb

INTERFACE
  SUBROUTINE print_days(month) &
      BIND(C, NAME='printDays')
  USE enumdefs
  INTEGER(KIND(jan)), VALUE :: month
  END SUBROUTINE print_days
END INTERFACE

  CALL print_days(jul)

  CALL print_days(feb)

END PROGRAM f_month
```

Running a slight variant of this code gives the output

```
month  7 has 31 days
month  2 has 28 or 29 days
```

Combining f_main and the C version of print_days given in (c_print_days.c) illustrates how enumeration constants interoperate.

```
#include<stdio.h>
enum months {jan=1, feb, mar, apr,
             may, jun, jul, aug,
             sep, oct, nov, dec};
void printDays(enum months month)
{
 switch (month) {
  case jan: case mar: case may:
  case jul: case aug: case oct:
  case dec:
  printf("month␣%2d␣has␣31␣days\n",
         month);
  break;
  case apr: case jun:
  case sep: case nov:
  printf("month␣%2d␣has␣30␣days\n",
         month);
  break;
  case feb:
  printf("month␣%2d␣has␣28/29␣days\n",
         month);
  break;
  default:
  printf("month␣%2d␣does␣not␣exist\n",
         month);
  break;
  }
}
```

Linking the Fortran version of print_days with the C driver program in (c_main.c) provides a second example.

```
enum months {jan=1, feb, mar, apr,
             may, jun, jul, aug,
             sep, oct, nov, dec};

void printDays(enum months month);

int main()
{
        printDays(jul);
        printDays(feb);

        return 0;
}
```

8. Other Functionality

We finish by mentioning a number of additional features and functionality that the iso_c_binding module provides that we have not mentioned in the preceding sections.

"Pointers" for C and Fortran are very different and are not interoperable with one another. A C pointer is just an address and, since it is considered to be a separate type, can be used to point to data of any type. On the other hand a Fortran pointer is defined to be of the same type as its target. Second, when considering array data, a Fortran pointer needs to store additional information regarding the bounds and shape of the array making it incompatible with C's concept of a simple address in memory. For example, it is not possible to make a non-contiguous array section the target of a C pointer.

However, non-contiguous array sections can still be passed as actual arguments to an interoperable C function since Fortran will perform copy-in and copy-out under these circumstances.

The iso_c_binding module provides two derived types to allow interoperability with C data pointers (c_ptr) and C function pointers (c_funptr). These may be used, for example, to preserve access to non-interoperable C data that needs to be preserved between calls to C functions. Accompanying these derived types are five subroutines and inquiry functions, (c_loc, c_funloc, c_associated, c_f_pointer, c_f_procpointer) that allow a variety of tests and conversions to be performed. For full details of the rules and restrictions governing these functions we refer the reader to (Adams et al., 2008, pp.569–579).

9. Closing Remarks

All the files mentioned here are available for testing, together with explanations on how to compile them, at
`http://www.nsc.liu.se/wg25/mlpcode/`.

We also tested our examples using Sun Fortran 95 8.2 (October 13, 2005). There is a minor error in this implementation; the standard requires

```
subroutine sub(f, b)  bind(c)
```

but this has to be written, using this version of the Sun compiler with a comma in order to avoid compilation errors

```
subroutine sub(f, b) , bind(c)
```

A similar remark holds for functions. This comma is optional in Sun Fortran 95 8.3 (July 18, 2007).

There is also a problem on some systems in using `CHARACTER(c_char)` since `CHARACTER(length)` has the same syntax. We therefore recommend using `CHARACTER(KIND=c_char)`.

The interoperability problem can be messy, even with the standard. For example, the enum construct (Metcalf et al., 2004, Section 14.11) is not implemented in all compilers. It is, for example, on IBM but not Intel 10.1*, Sun 8.3, and Portland pgf90 8.0-5†. Not only that, but Microsoft C++ does not have _Bool as a supported type. This makes portability difficult! If a programmer has to deal with problems like these then one may have to write two wrappers — in both languages to deal with non-portable data types and constructs, but the two wrappers can have a standard interface.

*It is confirmed that it is available on Intel 11.0.074.

†It is confirmed that it is available on Portland 9.0-2.

*References

Adams, J. C., Brainerd, W. S., Hendrickson, R. A., Maine, R. E., Martin, J. T., and Smith, B. T. (2008). *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*. Springer, London. ISBN 978-1-84628-378-9.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. SIAM, Philadelphia, third edition. ISBN 0-89871-447-8.

Barrachina, S., Castillo, M., Iqual, F. D., Mayo, R., and Quintana-Orti, E. S. (2008). Evaluation and tuning of the Level 3 CUBLAS for graphics processors. In *Proceedings of 22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008. Miami, Florida USA, April 14–18, 2008*, pages 3:1–8, IEEE, Los Alamitos, CA, USA. ISBN 978-1-4244-1693-6.

Chivers, I. D. and Sleightholme, J. (2009). Compiler support for the Fortran 2003 standard. *ACM SIGPLAN Fortran Forum*, 28(1):26–28. ISSN:1061-7264.

CUDA (2008). CUBLAS Library. Technical Report PG-00000-002_V2.0, NVIDIA, Santa Clara, CA 95050. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf`.

Dongarra, J., Du Croz, J., Duff, I., and Hammarling, S. (1990). A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17.

Einarsson, B. (1995). Mixed Language Programming, Part 4, Mixing ANSI-C with Fortran 77 or Fortran 90; Portability or Transportability? In *Current Directions in Numerical Software and High Performance Computing*, International Workshop, Kyoto, Japan. IFIP WG 2.5. `http://www.nsc.liu.se/~boein/ifip/kyoto/einarsson.html`.

Intel (2008). *Intel® Math Kernal Library*. Santa Clara, CA, USA, 630813-029US edition.

Metcalf, M., Reid, J., and Cohen, M. (2004). *Fortran 95/2003 explained*. Numerical Mathematics and Scientific Computation. Oxford University Press. ISBN 0-19-852693-8.