Interoperability with C in Fortran 2003

One of the major new features in the Fortran 2003 is features for interoperability with C (C Interop). The intrinsic module ISO_C_BINDING provides:

a) constants, mostly type parameters, C_NULL_CHAR, C_NULL_PTR, and others, b) types, and in particular, TYPE(C_PTR) and TYPE(C_FUNPTR), c) procedures, such as C_LOC, C_FUNLOC, C_F_POINTER, C_F_PROCPOINTER and C_ASSOCIATED.

A Fortran interface can be specified for a C function with external linkage and used to invoke such a function. The interface has the characteristic BIND(C) label, and must also satisfy some additional restrictions.

C Interop can be used to portably use multi-language codes in Fortran. Since most languages interoperate with C, the feature can actually be used to interoperate with other programming languages as well. C Interop can also be used to give access to Fortran programmers to the many standard libraries with widely-used and implemented C interfaces. This includes lower-level tasks such as interfacing with the OS on UNIX-based systems, or using special libraries like OpenGL.

For simple API's, developing Fortran interfaces is practically trivial once one gets some experience. For more complicated API's whose full functionality/power is not needed, such as for example TCP/IP sockets or shared-memory segments on UNIX systems, it is often easier to develop a condensed C API/library that does the actual work, and is simpler to interface to from Fortran. However, for libraries like OpenGL, one should provide a full Fortran interface so that the whole API can be accessed. Doing this manually is not easy and is also error-prone due to the size of the OpenGL/GLU/GLUT interfaces. For certain libraries like MPI, a special Fortran interface may be defined for the purposes of efficiency, portability, ease-of-use, or to accomodate for language semantic differences.

In this first paper, we will show how to develop a Fortran interface for a simple C API/library. In a second paper, we consider automating the process so that large and more complex API's, and in particular, OpenGL, can be handled. The source codes can be obtained at http://atom.princeton.edu/donev/F2x.

Along the way, we identify some problems with the design of C Interop in Fortran 2003. Many compilers have (partially) implemented C Interop. The example presented here also requires the use of procedure pointers, another new feature of Fortran 2003.

1. Introduction

We illustrate the use of C Interop with a simple but practically-useful example. Specifically, we use the standard UNIX header file `<dlfcn.h>` and the associated `<dl>` library (often `/usr/lib/libdl.so`) in order to invoke a procedure from a dynamic library (DL) (also called a shared library). Specifications for the `<dl>` library can be found at: http://www.opengroup.org/onlinepubs/007908799/xsh/dlfcn.h.html

DLs are usually linked with user programs at program startup by the dynamic linker. However, it is sometimes useful to dynamically link to a library at runtime. This can be used, for example, to upgrade pieces of in a running program. We provide an example of how to do this in a relatively portable way (at least to UNIX-like OS's) using C Interop.

We should note that lots of C API's specify that some functions may be implemented as macros (defined in the appropriate header file). If this is the case, one should write wrapper C functions that have external linkage and use the macros internally. In a lot of cases however, standard C libraries already provide functions with external linkage in the appropriate library, and I assume that this is the case for the `<dl>` library.

2. Fortran module for `<dlfcn.h>`

The interface for the `<dl>` library is particularly simple, and therefore we will use a manual approach to interface with it. First, we use a very simple C program to print out the values of the constants defined in the header file `<dlfcn.h>`:

```
! File dlfcn_c.c
#include <dlfcn.h>
main() {
   printf("RTLD_LAZY=%d, RTLD_NOW=%d, RTLD_GLOBAL=%d, RTLD_LOCAL=%d\n",\
       (int)RTLD_LAZY, (int)RTLD_NOW, (int)RTLD_GLOBAL, (int)RTLD_LOCAL);
}
```

The output from running this program can be used to declare Fortran PARAMETERs for these predefined constants:

```
> gcc -o dlfcn_c.x dlfcn_c.c
> dlfcn_c.x
RTLD_LAZY=1, RTLD_NOW=2, RTLD_GLOBAL=256, RTLD_LOCAL=0
```

A. Dealing with Null-Terminated Strings

The `<dl>` library has functions that return C strings (in this case error messages), which are simply pointers to a null-delimited character array. In Fortran character strings have length information associated with them, and therefore in order to use the strings returned by C functions in Fortran (for example, in I/O statements), we need a function that will convert a C pointer to a Fortran character array pointer of the appropriate length. We use the `<string.h>` function `<strlen>` to obtain the length of the null-terminated C string. The following module provides such a function:

```
! File dlfcn.f90
MODULE ISO_C_UTILITIES
   USE ISO_C_BINDING ! Intrinsic module

   CHARACTER(C_CHAR), DIMENSION(1), SAVE, TARGET, PRIVATE :: dummy_string="?"
```

```
CONTAINS

   FUNCTION C_F_STRING(CPTR) RESULT(FPTR)
      ! Convert a null-terminated C string into a Fortran character array pointer
      TYPE(C_PTR), INTENT(IN) :: CPTR ! The C address
      CHARACTER(KIND=C_CHAR), DIMENSION(:), POINTER :: FPTR

      INTERFACE ! strlen is a standard C function from <string.h>
         ! int strlen(char *string)
         FUNCTION strlen(string) RESULT(len) BIND(C,NAME="strlen")
            USE ISO_C_BINDING
            TYPE(C_PTR), VALUE :: string ! A C pointer
         END FUNCTION
      END INTERFACE

      IF(C_ASSOCIATED(CPTR)) THEN
         CALL C_F_POINTER(FPTR=FPTR, CPTR=CPTR, SHAPE=[strlen(CPTR)])
      ELSE
         ! To avoid segfaults, associate FPTR with a dummy target:
         FPTR=>dummy_string
      END IF

   END FUNCTION

END MODULE
```

B. Module DLFCN

We are now ready to write the module DLFCN. The function DLOpen opens a dynamic library (on UNIX systems, the LD_LIBRARY_PATH environmental variable can be used to specify the search path) and returns a handle for the library if successful and NULL otherwise. The function DLSym is then used to search for a procedure (symbol) with a specified name in the dynamic library, and to return a function pointer to it if successful or NULL otherwise. The procedure C_F_PROCPOINTER from the intrinsic module ISO_C_BINDING can be used to convert this to a Fortran procedure pointer, and thus to actually invoke the procedure. The function DLClose is used to close an open dynamic library, and DLError to obtain a string describing the last error that occured, if any.

Some comments are provided in the code itself, however, we hope the code is self-clarifying. The C prototypes from <dlfcn.h> are also given in a comments.

```
! File dlfcn.f90
MODULE DLFCN
   USE ISO_C_BINDING
   USE ISO_C_UTILITIES
   IMPLICIT NONE
   PRIVATE

   PUBLIC :: DLOpen, DLSym, DLClose, DLError ! DL API

   ! Valid modes for mode in DLOpen:
   INTEGER, PARAMETER, PUBLIC :: RTLD_LAZY=1, RTLD_NOW=2, RTLD_GLOBAL=256,
RTLD_LOCAL=0
```

```
      ! Obtained from the output of the previously listed C program

   INTERFACE ! All we need is interfaces for the prototypes in <dlfcn.h>
      FUNCTION DLOpen(file,mode) RESULT(handle) BIND(C,NAME="dlopen")
         ! void *dlopen(const char *file, int mode);
         USE ISO_C_BINDING
         CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: file
            ! C strings should be declared as character arrays
         INTEGER(C_INT), VALUE :: mode
         TYPE(C_PTR) :: handle
      END FUNCTION
      FUNCTION DLSym(handle,name) RESULT(funptr) BIND(C,NAME="dlsym")
         ! void *dlsym(void *handle, const char *name);
         USE ISO_C_BINDING
         TYPE(C_PTR), VALUE :: handle
         CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: name
         TYPE(C_FUNPTR) :: funptr ! A function pointer
      END FUNCTION
      FUNCTION DLClose(handle) RESULT(status) BIND(C,NAME="dlclose")
         ! int dlclose(void *handle);
         USE ISO_C_BINDING
         TYPE(C_PTR), VALUE :: handle
         INTEGER(C_INT) :: status
      END FUNCTION
      FUNCTION DLError() RESULT(error) BIND(C,NAME="dlerror")
         ! char *dlerror(void);
         USE ISO_C_BINDING
         TYPE(C_PTR) :: error
      END FUNCTION
   END INTERFACE

END MODULE
```

3. Example

We now provide an example of how to use the DLFCN module.

A. The shared library

First, we need to make a dynamic library. For this purpose, we can separately compile a simple subroutine using the appropriate compiler switches:

```
! File shared.f90
SUBROUTINE MySub(x) BIND(C,NAME="MySub")
   USE ISO_C_BINDING
   REAL(C_DOUBLE), VALUE :: x
   WRITE(*,*) "MySub: x=",x
END SUBROUTINE
```

We compile this file on a Linux machine with:

```
> f95 -c -pic shared.f90 -o shared.o
> ld -shared shared.o -o shared.so
```

B. Using the shared library

Note that Fortran 2003 allows one to pass a character string of any length as an actual argument corresponding to a character array, which is used below to simplify the passing of strings. It is important not to forget to NULL-terminate strings before passing them to C however: This is not done automatically!

```
! File dlfcn.f90
PROGRAM DLFCN_Test
   USE ISO_C_BINDING
   USE ISO_C_UTILITIES
   USE DLFCN
   IMPLICIT NONE

   ! Local variables:
   CHARACTER(KIND=C_CHAR,LEN=1024) :: dll_name, sub_name
   TYPE(C_PTR) :: handle=C_NULL_PTR
   TYPE(C_FUNPTR) :: funptr=C_NULL_FUNPTR
   INTEGER(C_INT) :: status

   ! The dynamic subroutine has a simple interface:
   ABSTRACT INTERFACE
      SUBROUTINE MySub(x) BIND(C)
         USE ISO_C_BINDING
         REAL(C_DOUBLE), VALUE :: x
      END SUBROUTINE
   END INTERFACE
   PROCEDURE(MySub), POINTER :: dll_sub ! Dynamically-linked procedure

   WRITE(*,*) "Enter the name of the DL and the name of the DL subroutine:"
   READ(*,"(A)") dll_name ! Enter "shared.so"
   READ(*,"(A)") sub_name ! Enter "MySub"

   ! Open the DL:
   handle=DLOpen(TRIM(dll_name)//C_NULL_CHAR, IOR(RTLD_NOW, RTLD_GLOBAL))
      ! The use of IOR is not really proper...wait till Fortran 2008
   IF(.NOT.C_ASSOCIATED(handle)) THEN
      WRITE(*,*) "Error in dlopen: ", C_F_STRING(DLError())
      STOP
   END IF

   ! Find the subroutine in the DL:
   funptr=DLSym(handle,TRIM(sub_name)//C_NULL_CHAR)
   IF(.NOT.C_ASSOCIATED(funptr)) THEN
      WRITE(*,*) "Error in dlsym: ", C_F_STRING(DLError())
      STOP
   END IF
   ! Now convert the C function pointer to a Fortran procedure pointer
   CALL C_F_PROCPOINTER(CPTR=funptr, FPTR=dll_sub)
   ! Finally, invoke the dynamically-linked subroutine:
   CALL dll_sub(1.0_c_double)
   ! Now close the DL:
   status=DLClose(handle)
   IF(status/=0) THEN
      WRITE(*,*) "Error in dlclose: ", C_F_STRING(DLError())
      STOP
   END IF

END PROGRAM
```

We can now compile and run this program:

```
> f95 dlfcn.f90 -o dlfcn.x -ldl
> dlfcn.x
 Enter the name of the DLL and the name of the DLL subroutine:
shared.so
MySub
 MySub: x=   1.0000000000000000
```

We invite the readers to test this code for themselves, and try writing some BIND(C) interfaces of their own!