Physics 555B: Mixed Language (Fortran and C) Programming

Please report all errors/typos. etc to choptuik@physics.ubc.ca

Last updated January 2007

From time to time it may prove convenient and/or necessary to call a Fortran 77 subroutine or function from C source code, or, conversely, to call a C function from some Fortran 77 source code. Due to Fortran 77's extremely limited set of data types (essentially scalars and 1-dimensional arrays of integer, real and logical variables (perhaps of various numbers of bytes), as well as character strings), it is almost always possible to call Fortran routines from C. On the other hand, due to the richer variety of types in C, including user-defined types, structures and pointer-types, it is *not* always possible to call C routines from Fortran. In these notes we will discuss how to mix and match C and Fortran code in a typical Unix environment such as Linux. As was the case in our discussion of the basic use of Fortran and C under Unix, for concreteness we will use the **PG** compilers for our illustrative examples. However, the same mechanisms will work with most other compiler suites typically used on current Unix systems, with a few provisos as noted.

Three key observations/facts underly the task of mixed Fortran 77 (hereafter simply Fortran) and C programming:

1. As already mentioned, C routines which are to be called from Fortran must have formal parameters (arguments) whose types are among the relatively few types available in Fortran: namely, scalars and 1-dimensional arrays (or vectors) of integer, real and logical variables and character strings. Since we are primarily concerned with scientifc programming, we will make the further restriction to formal parameters that are of one of the following two types (scalars or 1-dimensional arrays):
    1. 4-byte integers: i.e. **integer** variables in Fortran, **int** variables in C
    2. 8-byte reals: i.e. **real*8** variables in Fortran, **double** variables in C

2. C implements so-called **call-by-value** for passing actual **scalar** arguments to routines (C functions). This fact will be illustrated with a simple example below. Arguments that are **arrays**, on the other hand, have their *addresses* passed in C (**call-by-address**). In Fortran, *all* arguments, be they scalars or arrays are passed by address, one consequence of which is the fact that from the point of view of passing arguments to a Fortran subroutine or function, there is no distinction between a scalar variable and a 1-dimensional array variable that has length 1. Again, the fact that Fortran is always call-by-address will be illustrated in an example to follow.

3. As discussed previously, in order that routines written in different languages, but using the same routine name in their respective source codes, be distinguishable at the "system level", i.e. at load time, some mechanism must exist to transform ("mung") the routine names to global symbols for all but a single language. In Unix, this single language is naturally C, since Unix implementations (including Linux) tend to be coded primarily in C.

   For the case of Fortran, most compiler vendors, including the Portland Group, have adopted a convention first introduced by Sun Microsystems, wherein Fortran routine names are translated to external names by appending a single underscore (_) to the source code name. A notable exception in this instance are the **GNU** Fortran compilers, which by default (apparently for sheer perversity), append a single underscore to a Fortran routine name, unless said name already contains an underscore, in which case *two* underscores are appended! Fortunately, there is an option to the **f77**, namely **-fno-second-underscore**, that instructs the compiler to act in a sane and conventional fashion, and use of this switch is recommended practice.

Call-by-value versus call-by-address

We will first consider two simple programs that illustrate the distinction between the call-by-value and call-by-address mechanisms for scalar arguments in C and Fortran code respectively.

Here is a simple C program consisting of a main routine and a single function that is invoked by the main routine:

```
#include <stdio.h>
/*============================================================
  Simple example to illustrate fact that scalar parameters
  are passed BY VALUE in C.  Calling function 'c_fcn' has
  NO side effect vis a vis the value of 'i' in the main
  routine.
============================================================*/
void c_fcn(int ival);

int main(int argc, char ** argv) {
   int   i;

   i = 10;
   printf("c_call: In main before call to c_fcn, i = %d\n\n",i);
   (void) c_fcn(i);
   printf("c_call: In main after call to c_fcn, i = %d\n",i);
```

```
}

void c_fcn(int ival) {
   printf("c_fcn: Entering routine, ival = %d\n",ival);
   ival = 0;
   printf("c_fcn: Exiting routine, ival = %d\n\n",ival);
}
```

With an appropriate Makefile, we can build the corresponding executable in the usual fashion:

```
% make c-call

pgcc -g -c c-call.c
pgcc -g -L/usr/local/PGI/lib c-call.o -o c-call
```

Execution of the program then produces output as follows:

```
% c_call

c_call: In main before call to c_fcn, i = 10

c_fcn: Entering routine, ival = 10
c_fcn: Exiting routine, ival = 0

c_call: In main after call to c_fcn, i = 10
```

The key thing to observe here is that the assignment statement

```
   ival = 0;
```

in the function **c_fcn** has no effect on the value of the corresponding actual argument, **i**, in the main program. In effect, when the function is called from the main program, the function allocates *new* storage for the variable **ival** and initializes that storage by *copying the value* of **i** from the main program.

Contrast the above behaviour with that of the analogous Fortran program:

```
c============================================================
c     f77_call:
c
c     Simple example to illustrate fact that ALL routine
c     parameters, including scalars, are passed BY
c     ADDRESS in f77.  Calling subroutine 'f77_sub' has
c     side-effect of setting the value of 'i' in the
c     main routine to 0.
c============================================================
      program     f77_call

      implicit    none

      integer     i

      i = 10
      write(*,*) 'f77_call: In main before call to f77_sub,',
     &           ' i = ', i
      write(*,*)
      call f77_sub(i)
      write(*,*) 'f77_call: In main after call to f77_sub,',
     &           ' i = ', i

      stop

      end

      subroutine  f77_sub(ival)
         implicit  none

         integer   ival

         write(*,*) 'f77_sub: Entering routine, ival = ', ival
```

```
         ival = 0
         write(*,*) 'f77_sub: Exiting routine, ival = ', ival
         write(*,*)

         return
      end
```

Again, creation of the corresponding executable is straightforward:

```
% make f77-call

pgf77 -g -c f77-call.f
pgf77 -g -L/usr/local/PGI/lib f77-call.o -o f77-call
```

and execution of **f77-call** produces the following output:

```
% f77-call

 f77_call: In main before call to f77_sub, i =           10

 f77_sub: Entering routine, ival =           10
 f77_sub: Exiting routine, ival =            0

 f77_call: In main after call to f77_sub, i =            0
```

In this case the assignment statement

```
         ival = 0
```

in the **f77_sub** subroutine *does* change the value of the variable, **i**, that is supplied as an argument to **f77_sub** in the main program. Indeed, because of the call-by-address mechanism, **i** in the main program and **ival** in the subroutine *occupy, during the execution of the subroutine, the same storage.*

Calling Fortran routines from C

Bearing in mind the three key observations enumerated above, calling Fortran routines from C is quite straightforward, if somewhat cumbersome. Here are the contents of a source file, **f77-routines.f**, containing two functions and one subroutine:

```
c=============================================================
c     Definitions of f77 routines that are to be called from
c     C.
c=============================================================

      integer function f77_int_fcn(isc, iarray, liarray)
         implicit    none

         integer     isc, liarray, iarray(liarray)
         integer     i

         write(*,*) 'f77_int_fcn: Entering routine'
         write(*,*) ' isc = ', isc
         do i = 1 , liarray
            write(*,*) ' iarray(', i, ') = ', iarray(i)
         end do
         write(*,*) 'f77_int_fcn: Exiting routine'
         write(*,*)

         f77_int_fcn = -1

         return

      end

      real*8 function f77_real8_fcn(rsc, rarray, lrarray)
         implicit    none

         integer     lrarray
         real*8      rsc, rarray(lrarray)
         integer     i
```

```
         write(*,*) 'f77_real8_fcn: Entering routine'
         write(*,*) ' rsc = ', rsc
         do i = 1 , lrarray
            write(*,*) ' rarray(', i, ') = ', rarray(i)
         end do
         write(*,*) 'f77_real8_fcn: Exiting routine'
         write(*,*)

         f77_real8_fcn = -1.0d0

         return

      end

      subroutine f77_sub(isc, rsc)
         implicit    none

         integer     isc
         real*8      rsc

         write(*,*) 'f77_sub: Entering routine'
         write(*,*) ' isc = ', isc
         write(*,*) ' rsc = ', rsc
         write(*,*) 'f77_sub: Exiting routine'
         write(*,*)

         return
      end
```

and here is the C main routine, contained in a source file, **c-calls-f77.c**, that calls the Fortran functions and subroutine:

```
#include <stdio.h>
/*============================================================
   c_calls_f77

   Illustrates invocation of f77 routines from C program
   assuming that the compiler system uses "append single
   underscore" conventiopn for translating f77 names to
   external symbols.

   f77 routines are defined in source file 'f77-routines.f'
============================================================*/

/*------------------------------------------------------------
   Declarations of f77 routines.
------------------------------------------------------------*/
int     f77_int_fcn_(int * pisc, int * iarray, int * pliarray);
double  f77_real8_fcn_(double * prsc, double * rarray, int * plrarray);
void    f77_sub_(int * pisc, double * prsc);

int main(int argc, char ** argv) {

/*------------------------------------------------------------
   Variables to be passed to f77 routines.
------------------------------------------------------------*/
   int     isc,  liarray,  lrarray,  iarray[10];
   double  rsc,  rarray[10];

/*------------------------------------------------------------
   Miscellaneous variables.
------------------------------------------------------------*/
   int     i,    ival;
   double  rval;

   isc = 0;
   rsc = 0;
   liarray = 10;
   lrarray = 10;
```

```
   for( i = 0; i < liarray; i++ ) {
      iarray[i] = i;
      rarray[i] = i;
   }

/*-------------------------------------------------------
   Invoke f77 routines.  Note that ADDRESSES of scalar vbls
   must be passed for those parameters where the f77 routine
   expects a scalar.
-------------------------------------------------------*/
   ival = f77_int_fcn_(&isc, iarray, &liarray);
   rval = f77_real8_fcn_(&rsc, rarray, &liarray);
   (void) f77_sub_(&isc, &rsc);

   printf("f77_int_fcn(...) returns %d\n",ival);
   printf("f77_real8_fcn(...) returns %g\n",rval);

   exit(0);
}
```

Note that the declarations (prototypes) of the Fortran functions and subroutines must precede the invocations of these routines by the C-code, and that the Fortran routines must be declared and invoked with a trailing underscore, per the discussion above. Also note how the **addresses** of scalar variables such as **isc** and **rsc** must be supplied to the Fortran routines via the "address-of" operator, **&**.

To build the corresponding executable, each of the two source files, **c-calls-f77.c** and **f77-routines.f**, must be compiled with the appropriate compiler, using the **-c** option in each case to produce object code. The two object files, **c-calls-f77.o** and **f77-routines.o** are then linked together using the C compiler, which in turn invokes the loader program, **ld**.

Crucially, in the link phase, additional libraries which provide the basic run time support for *Fortran* programs must be supplied to the loader. Which specific libraries must be specified will *always* depend on which compiler suite is being used, and often on which specific version of a given suite is employed. One must often scour manuals and user guides to determine the correct libraries, although in the current epoch, the needed information can also frequently be found via on-line searches.

For the version of the **PG** compilers currently being used in this course, only a single library, **libpgftnftl.a**, must be included to provide the Fortran run time support, so we only need to ensure that

```
   -lpgftnrtl
```

is included in the link command.

The executable for our C-calls-Fortran example can thus be generated as follows

```
% make c-calls-f77

pgcc -g -c c-calls-f77.c
pgf77 -g -c f77-routines.f
pgcc -g -L/usr/local/PGI/lib c-calls-f77.o f77-routines.o -lpgftnrtl \
   -o c-calls-f77
```

Note that if we had omitted **-lpgftnrtl**, in the load phase, the linker would have complained vociferously and voluminously:

```
% pgcc -g -L/usr/local/PGI/lib c-calls-f77.o f77-routines.o -o c-calls-f77

f77-routines.o(.text+0x3c): In function ‘f77_int_fcn’:
f77-routines.f:12: undefined reference to ‘fio_src_info’
f77-routines.o(.text+0x55):f77-routines.f:12: undefined reference to ‘fio_ldw_init’
f77-routines.o(.text+0x83):f77-routines.f:12: undefined reference to ‘fio_ldw’
                                    .
                                    .
                                    .
f77-routines.o(.text+0x2e3):f77-routines.f:18: undefined reference to ‘fio_ldw_end’
f77-routines.o(.text+0x32c): In function ‘f77_real8_fcn’:
f77-routines.f:33: undefined reference to ‘fio_src_info’
f77-routines.o(.text+0x345):f77-routines.f:33: undefined reference to ‘fio_ldw_init’
f77-routines.o(.text+0x373):f77-routines.f:33: undefined reference to ‘fio_ldw’
                                    .
                                    .
                                    .
```

```
f77-routines.o(.text+0x5d3):f77-routines.f:39: undefined reference to ‘fio_ldw_end’
f77-routines.o(.text+0x611): In function ‘f77_sub’:
f77-routines.f:53: undefined reference to ‘fio_src_info’
f77-routines.o(.text+0x62a):f77-routines.f:53: undefined reference to ‘fio_ldw_init’
f77-routines.o(.text+0x658):f77-routines.f:53: undefined reference to ‘fio_ldw’
                                    .
                                    .
                                    .
f77-routines.o(.text+0x825):f77-routines.f:57: undefined reference to ‘fio_ldw_end’
```

The execution of **c-calls-f77** now produces the expected output:

```
% c-calls-f77

 f77_int_fcn: Entering routine
  isc =            0
  iarray(           1) =           0
  iarray(           2) =           1
  iarray(           3) =           2
  iarray(           4) =           3
  iarray(           5) =           4
  iarray(           6) =           5
  iarray(           7) =           6
  iarray(           8) =           7
  iarray(           9) =           8
  iarray(          10) =           9
 f77_int_fcn: Exiting routine

 f77_real8_fcn: Entering routine
  rsc =   0.0000000000000000E+000
  rarray(           1) =   0.0000000000000000E+000
  rarray(           2) =    1.000000000000000
  rarray(           3) =    2.000000000000000
  rarray(           4) =    3.000000000000000
  rarray(           5) =    4.000000000000000
  rarray(           6) =    5.000000000000000
  rarray(           7) =    6.000000000000000
  rarray(           8) =    7.000000000000000
  rarray(           9) =    8.000000000000000
  rarray(          10) =    9.000000000000000
 f77_real8_fcn: Exiting routine

 f77_sub: Entering routine
  isc =            0
  rsc =   0.0000000000000000E+000
 f77_sub: Exiting routine

f77_int_fcn(...) returns -1
f77_real8_fcn(...) returns -1
```

The above code can be used as template/model for any situation likely to be encountered in this course wherein a C-programmer wishes to call one or more Fortran routines from his/her C code.

Also, for convenience, here are the required Fortran run-time-support libraries currently needed for various compiler suites:

1. GNU compilers: **-lg2c**
2. PG compilers: **-lpgftnrtl**
3. Intel compilers, version 8 and later: **-lsvml -lifcore**

Calling C routines from Fortran

The task of calling C routines from Fortran is a little more tricky. Apart from the restriction that all of the formal arguments of the C routine to be called must be compatible with the quite restricted set of Fortran data types, it should be clear by now that only C routines with names that end with an underscore can be invoked directly from Fortran code.

Thus, in order to call a C routine having Fortran-compatible arguments, but that has a name that does *not* end in an underscore, we must add an additional layer of C code, which defines an "interface" routine, with a name that *does* end in an underscore, and which then invokes the desired C "base" routine.

The source code file **c-routines.c** defines both "interface" and "base" routines completely analogous to those defined

previously in **f-routines.f**:

```
#include <stdio.h>
#include "c-routines.h"

/*================================================================
   Definitions of interface routines that can be invoked
   DIRECTLY from Fortran.

   Note that since all arguments in f77 are passed BY ADDRESS,
   arguments that are scalar in the f77 call must be defined as
   pointer-to-appropriate-type in the code for the C function.
================================================================*/

int c_int_fcn_(int * pisc, int * iarray, int * pliarray) {
   return c_int_fcn(*pisc, iarray, *pliarray);
}

double c_double_fcn_(double * prsc, double * rarray, int * plrarray) {
   return c_double_fcn(*prsc, rarray, *plrarray);
}

void c_void_fcn_(int * pisc,double * prsc) {
   (void) c_void_fcn(*pisc, *prsc);
   return;
}

/*================================================================
   Definitions of C routines that are to be called (indirectly)
   from Fortran
================================================================*/

int c_int_fcn(int isc, int * iarray, int liarray) {
   int   i;

   printf("c_int_fcn: Entering routine ... \n");
   printf(" isc = %d\n",isc);
   printf(" liarray = %d\n",liarray);
   for( i = 0; i < liarray; i++) {
      printf("iarray[%d] = %d\n", i, iarray[i]);
   }
   printf("c_int_fcn: Exiting routine ... \n\n");

   return -1;
}

double c_double_fcn(double rsc, double * rarray, int lrarray) {
   int   i;

   printf("c_double_fcn: Entering routine ... \n");
   printf(" rsc = %d\n",rsc);
   printf(" liarray = %d\n",lrarray);
   for( i = 0; i < lrarray; i++) {
      printf("rarray[%d] = %g\n", i, rarray[i]);
   }
   printf("c_double_fcn: Exiting routine ... \n\n");

   return -1.0;
}

void c_void_fcn(int isc,double rsc) {
   printf("c_void_fcn: Entering routine ... \n");
   printf(" isc = %d\n",isc);
   printf(" rsc = %d\n",rsc);
   printf("c_void_fcn: Exiting routine ... \n\n");
}
```

Observe that there is one interface function for each base function, and that the interface function simply returns the invocation of the base function with scalar arguments dereferenced using the dereferencing operator, *****, as necessary.

The Fortran main program, **f77-calls-c.f** that invokes the C functions is as follows

```
c=============================================================
c     f77_calls_c:
c
c     Illustrates invocation of C routines from f77 program
c     assuming that the compiler system uses "append single
c     underscore" convention for translating f77 names to
c     external symbols.
c
c     C routines are defined in source file 'c-routines.c'.
c=============================================================
      program     f77_calls_c

      implicit    none

c-------------------------------------------------------------
c     Declaration of C-callable functions, no declarations
c     are necessary/possible for 'void' functions.
c     C routines per se must have '_' appended to the
c     function name in definition (see 'c-routines.c').
c-------------------------------------------------------------
      integer     c_int_fcn
      real*8      c_double_fcn

c-------------------------------------------------------------
c     Variables to be passed to C routines.
c-------------------------------------------------------------
      integer     isc,  liarray,  lrarray, iarray(10)
      real*8      rsc,  rarray(10)

c-------------------------------------------------------------
c     Miscellaneous variables.
c-------------------------------------------------------------
      integer     i,    ival
      real*8      rval

      isc = 0
      rsc = 0
      liarray = 10
      lrarray = 10
      do i = 1 , 10
         iarray(i) = i
         rarray(i) = i
      end do

c-------------------------------------------------------------
c     Corresponding C routines must have '_' appended to
c     function name: see 'c-routines.c'
c-------------------------------------------------------------
      ival = c_int_fcn(isc,iarray,liarray)
      rval = c_double_fcn(rsc,rarray,lrarray)
      call c_void_fcn(isc,rsc)

      write(*,*) 'c_int_fcn(...) returns', ival
      write(*,*) 'c_double_fcn(...) returns', rval

      stop
      end
```

Once again, in order to create the executable **f77-calls-c**, the two source files, **f77-calls-c.f** and **c-routines.c**, must be separately compiled with the Fortran and C compilers respectively, using the **-c** option to produce object code. However, in this case, when linking the resulting object files together to produce an executable with the Fortran compiler, there is no need to specify additional libraries, unless those libraries would be needed by the C routines themselves.

The executable is thus created as follows:

```
% make f77-calls-c
```

```
pgf77 -g -c f77-calls-c.f
pgcc -g -c c-routines.c
pgf77 -g -L/usr/local/PGI/lib f77-calls-c.o c-routines.o -o f77-calls-c
```

and, again, execution of **f77-calls-c** produces the expected output:

```
% f77-calls-c

c_int_fcn: Entering routine ...
 isc = 0
 liarray = 10
iarray[0] = 1
iarray[1] = 2
iarray[2] = 3
iarray[3] = 4
iarray[4] = 5
iarray[5] = 6
iarray[6] = 7
iarray[7] = 8
iarray[8] = 9
iarray[9] = 10
c_int_fcn: Exiting routine ...

c_double_fcn: Entering routine ...
 rsc = 0
 liarray = 10
rarray[0] = 1
rarray[1] = 2
rarray[2] = 3
rarray[3] = 4
rarray[4] = 5
rarray[5] = 6
rarray[6] = 7
rarray[7] = 8
rarray[8] = 9
rarray[9] = 10
c_double_fcn: Exiting routine ...

c_void_fcn: Entering routine ...
 isc = 0
 rsc = 0
c_void_fcn: Exiting routine ...

 c_int_fcn(...) returns          -1
 c_double_fcn(...) returns  -1.000000000000000
```