Chapter 38

C/Fortran interoperability

Most of the time, a program is written is written in a single language, but in some circumstances it is necessary or desirable to mix sources in more than one language for a single executable. One such case is when a library is written in one language, but used by a program in another. In such a case, the library writer will probably have made it easy for you to use the library; this section is for the case that you find yourself in the place of the library writer. We will focus on the common case of *interoperability* between C/C++ and Fortran.

This issue is complicated by the fact that both languages have been around for a long time, and various recent language standards have introduced mechanisms to facilitate interoperability. However, there is still a lot of old code around, and not all compilers support the latest standards. Therefore, we discuss both the old and the new solutions.

38.1 Linker conventions

As explained above, a compiler turns a source file into a binary, which no longer has any trace of the source language: it contains in effect functions in machine language. The linker will then match up calls and definitions, which can be in different files. The problem with using multiple languages is then that compilers have different notions of how to translate function names from the source file to the binary file.

Let's look at codes (you can find example files in `tutorials/linking`):

```
// C:
      Subroutine foo()
      Return
      End Subroutine
! Fortran
void foo() {
  return;
}
```

After compilation you can use *nm* to investigate the binary *object file*:

```
%% nm fprog.o
0000000000000000 T _foo_
....
%% nm cprog.o
0000000000000000 T _foo
....
```

You see that internally the `foo` routine has different names: the Fortran name has an underscore appended. This makes it hard to call a Fortran routine from C, or vice versa. The possible name mismatches are:

- The Fortran compiler appends an underscore. This is the most common case.
- Sometimes it can append two underscores.
- Typically the routine name is lowercase in the object file, but uppercase is a possibility too.

Since C is a popular language to write libraries in, this means that the problem is often solved in the C library by:

- Appending an underscore to all C function names; or
- Including a simple wrapper call:

```
int SomeCFunction(int i,float f)
{
  .....
}
int SomeCFunction_(int i,float f)
{
  return SomeCFunction(i,f);
}
```

38.1.1 C bindings in Fortran 2003

With the latest Fortran standard there are explicit *C bindings*, making it possible to declare the external name of variables and routines:

```
module operator
  real, bind(C) :: x
contains
  subroutine s() bind(C,name='s')
  return
  end subroutine
end module

%% ifort -c fbind.F90
%% nm fbind.o
.... T _s
.... C _x
```

It is also possible to declare data types to be C-compatible:

```
Program fdata

  use iso_c_binding

  type, bind(C) :: c_comp
    real (c_float)  :: data
    integer (c_int) :: i
    type (c_ptr)    :: ptr
  end type

end Program fdata
```

38.1.2 C++ linking

Libraries written in C++ offer further problems. The C++ compiler makes external symbols by combining the names a class and its methods, in a process known as *name mangling*. You can force the compiler to generate names that are intelligible to other languages by

```
#ifdef __cplusplus
  extern"C" {
#endif
  .
  .
  place declarations here
  .
  .
#ifdef __cplusplus
  }
#endif
```

Example: compiling

```
#include <stdlib.h>

int foo(int x) {
  return x;
}
```

and inspecting the output with `nm` gives:

```
0000000000000010 s EH_frame1
0000000000000000 T _foo
```

On the other hand, the identical program compiled as C++ gives

```
0000000000000010 s EH_frame1
0000000000000000 T __Z3fooi
```

You see that the name for `foo` is something mangled, so you can not call this routine from a program in a different language. On the other hand, if you add the `extern` declaration:

```
#include <stdlib.h>

#ifdef __cplusplus
  extern"C" {
#endif
int foo(int x) {
  return x;
}
#ifdef __cplusplus
  }
#endif
```

you again get the same linker symbols as for C, so that the routine can be called from both C and Fortran.

If your main program is in C, you can use the C++ compiler as linker. If the main program is in Fortran, you need to use the Fortran compiler as linker. It is then necessary to link in extra libraries for the C++ system routines. For instance, with the Intel compiler `-lstdc++ -lc` needs to be added to the link line.

The use of `extern` is also needed if you link other languages to a C++ main program. For instance, a Fortran subprogram `foo` should be declared as

```
extern "C" {
void foo_();
}
```

In that case, you again use the C++ compiler as linker.

38.1.3 Complex numbers

The *complex data types in C/C++ and Fortran* are compatible with each other. Here is an example of a C++ program linking to Lapack's complex vector scaling routine `zscal`.

```
// zscale.cxx
extern "C" {
void zscal_(int*,double complex*,double complex*,int*);
}
  complex double *xarray,*yarray, scale=2.;
  xarray = new double complex[n]; yarray = new double complex[n];
  zscal_(&n,&scale,xarray,&ione);
```

38.2 Arrays

C and Fortran have different conventions for storing multi-dimensional arrays. You need to be aware of this when you pass an array between routines written in different languages.

Fortran stores multi-dimensional arrays in *column-major* order; see figure 38.1. For two dimensional arrays `A(i,j)` this means that the elements in each column are stored contiguously: a 2×2 array is stored as `A(1,1), A(2,1), A(1,2), A(2,2)`. Three and higher dimensional arrays are an obvious exten-

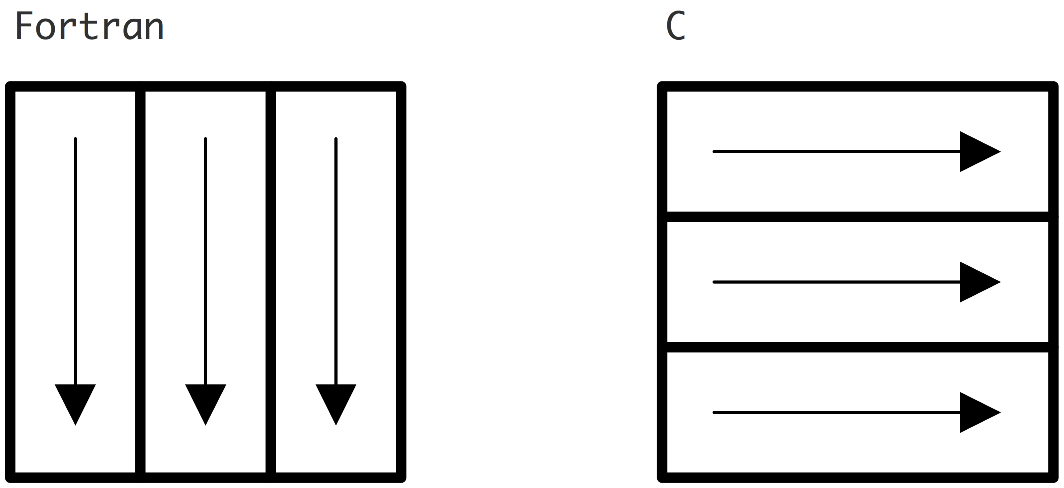

Figure 38.1: Fortran and C array storage by columns and rows respectively

sion: it is sometimes said that 'the left index varies quickest'.

C arrays are stored in *row-major* order: elements in each row are stored contiguous, and columns are then placed sequentially in memory. A 2×2 array `A[2][2]` is stored as `A[1][1], A[1][2], A[2][1], A[2][2]`.

A number of remarks about arrays in C.

- C (before the C99 standard) has multi-dimensional arrays only in a limited sense. You can declare them, but if you pass them to another C function, they no longer look multi-dimensional: they have become plain `float*` (or whatever type) arrays. That brings us to the next point.
- Multi-dimensional arrays in C look as if they have type `float**`, that is, an array of pointers that point to (separately allocated) arrays for the rows. While you could certainly implement this:

  ```
  float **A;
  A = (float**)malloc(m*sizeof(float*));
  for (i=0; i<n; i++)
    A[i] = (float*)malloc(n*sizeof(float));
  ```

  careful reading of the standard reveals that a multi-dimensional array is in fact a single block of memory, no further pointers involved.

Given the above limitation on passing multi-dimensional arrays, and the fact that a C routine can not tell whether it's called from Fortran or C, it is best not to bother with multi-dimensional arrays in C, and to emulate them:

```
float *A;
A = (float*)malloc(m*n*sizeof(float));
#define SUB(i,j,m,n) i+j*m
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    .... A[SUB(i,j,m,n)] ....
```

where for interoperability we store the elements in column-major fashion.

38.2.1 Array alignment

For reasons such as SIMD *vector instructions*, it can be advantageous to use *aligned allocation*. For instance, '16-byte alignment' means that the starting address of your array, expressed in bytes, is a multiple of 16.

In C, you can force such alignment with `posix_memalign`. In Fortran there is no general mechanism for this. The Intel compiler allows you to write:

```
double precision, allocatable :: A(:), B(:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A, B
```

38.3 Strings

Programming languages differ widely in how they handle strings.

- In C, a string is an array of characters; the end of the string is indicated by a null character, that is the ascii character zero, which has an all zero bit pattern. This is called *null termination*.
- In Fortran, a string is an array of characters. The length is maintained in a internal variable, which is passed as a hidden parameter to subroutines.
- In Pascal, a string is an array with an integer denoting the length in the first position. Since only one byte is used for this, strings can not be longer than 255 characters in Pascal.

As you can see, passing strings between different languages is fraught with peril. This situation is made even worse by the fact that passing strings as subroutine arguments is not standard.

Example: the main program in Fortran passes a string

```
Program Fstring
  character(len=5) :: word = "Word"
  call cstring(word)
end Program Fstring
```

and the C routine accepts a character string and its length:

```
#include <stdlib.h>
#include <stdio.h>

void cstring_(char *txt,int txtlen) {
  printf("length = %d\n",txtlen);
  printf("<<");
  for (int i=0; i<txtlen; i++)
    printf("%c",txt[i]);
  printf(">>\n");
}
```

which produces:

```
length = 5
<<Word >>
```

To pass a Fortran string to a C program you need to append a null character:

```
call cfunction ('A string'//CHAR(0))
```

Some compilers support extensions to facilitate this, for instance writing

```
DATA forstring /'This is a null-terminated string.'C/
```

Recently, the 'C/Fortran interoperability standard' has provided a systematic solution to this.

38.4 Subprogram arguments

In C, you pass a `float` argument to a function if the function needs its value, and `float*` if the function has to modify the value of the variable in the calling environment. Fortran has no such distinction: every variable is passed *by reference*. This has some strange consequences: if you pass a literal value `37` to a subroutine, the compiler will allocate a nameless variable with that value, and pass the address of it, rather than the value[1].

For interfacing Fortran and C routines, this means that a Fortran routine looks to a C program like all its argument are 'star' arguments. Conversely, if you want a C subprogram to be callable from Fortran, all its arguments have to be star-this or that. This means on the one hand that you will sometimes pass a variable by reference that you would like to pass by value.

Worse, it means that C subprograms like

```
void mysub(int **iarray) {
 *iarray = (int*)malloc(8*sizeof(int));
 return;
```

1. With a bit of cleverness and the right compiler, you can have a program that says `print *,7` and prints `8` because of this.

```
    }
```

can not be called from Fortran. There is a hack to get around this (check out the Fortran77 interface to the Petsc routine `VecGetValues`) and with more cleverness you can use `POINTER` variables for this.

38.5 Input/output

Both languages have their own system for handling input/output, and it is not really possible to meet in the middle. Basically, if Fortran routines do I/O, the main program has to be in Fortran. Consequently, it is best to isolate I/O as much as possible, and use C for I/O in mixed language programming.

38.6 Fortran/C interoperability in Fortran2003

The latest version of Fortran, unsupported by many compilers at this time, has mechanisms for interfacing to C.

- There is a module that contains named kinds, so that one can declare
  ```
  INTEGER,KIND(C_SHORT) :: i
  ```
- Fortran pointers are more complicated objects, so passing them to C is hard; Fortran2003 has a mechanism to deal with C pointers, which are just addresses.
- Fortran derived types can be made compatible with C structures.

38.7 Python calling C code

Because of its efficiency of computing, C is a logical language to use for the lowest layers of a program. On the other hand, because of its expressiveness, Python is a good candidate for the top layers. It is then a logical thought to want to call C routines from a python program. This is possible using the python *ctypes* module.

1. You write your C code, and compile it to a dynamic library as indicated above;
2. The python code loads the library dynamically, for instance for `libc`:
   ```
   path_libc = ctypes.util.find_library("c")
   libc = ctypes.CDLL(path_libc)
   libc.printf(b"%s\n", b"Using the C printf function from Python ... ")
   ```
3. You need to declare what the types are of the C routines in python:
   ```
   test_add = mylib.test_add
   test_add.argtypes = [ctypes.c_float, ctypes.c_float]
   test_add.restype = ctypes.c_float
   test_passing_array = mylib.test_passing_array
   test_passing_array.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c
   test_passing_array.restype = None
   ```

4. Scalars can be passed simply; arrays need to be constructed:
   ```
   data = (ctypes.c_int * Nelements)(*[x for x in range(numel)])
   ```