Programming

Programming with Fortran and C

- Style and rules
- Statements
- Examples
- Compilers and options

Before we start coding:

- Check formulae and algorithms. It takes very long time to develop and to debug a code. If at the end we find that there was a mistake in equations, we lose too much.
- More efficient algorithm has higher priority over pretty code writing.
- Start with clearly defining data structures: arrays, parameters, files.
- Estimate time needed for code developing: 10% - design, 10% - write, 80% - debug
- High price for making errors: write code nice and clear; learn how find bugs.

Style and rules

- Write code starting with defining objects and data flow.
- Top down design: start with the top level of the code. Write the logic of calls of different top-level routines. Then fill the gaps by writing routines of lower and lower levels.
- Routines should be short. No side effects. If a routine is assigning density of particles to some array, do not modify potential.

More on Style: calling names

- Use long and natural names for *global variables and arrays*. E.g., *Nparticles*, *Niterations*.
- For local temporary variables use short names: *x,y,i,k0*
- In a long subroutine one should be careful not to use names, which are already in use. You may give names, which are obviously temporary: *tmp, dummy*
- Use ‘grab’ to check whether name is used.

Example:

assign values to a 3D matrix and get the sum of squares of all elements

Declarations:

Main body:

Finish:

```
!-------------------------------------------------------------
!                              test memory: fortran-77 style
!-------------------------------------------------------------
      parameter (N =50 )
      COMMON /aa/ A(N,N,N)              ! allocate memory for array A
      real*8    sum

      write (*,*)  ' Required Memory=',4.*N**3/1024.**2,'Mb'
      sum = 0.
      Do k=1,N
      Do j=1,N
         Do i=1,N
            A(i,j,k) = ((i+j+k)/100.)    ! just an example
            sum =sum + A(i,j,k)**2       ! make sum of squares
         EndDo
      EndDo
      EndDo

      write (*,*)  ' rms=',sqrt(sum/N**3)

      stop
      end
```

The same example in Fortran-90 and C

Declarations:

Main body:

Finish:

```
!---------------------------------------------------------------
!                            test memory: fortran-90 style
!---------------------------------------------------------------
Integer, parameter        :: N=50
real, allocatable         :: A(:,:,:)
real*8                    :: summ =0

write (*,*)  ' Required Memory=',4.*N**3/1024.**2,'Mb'

allocate (A(N,N,N))       ! this allocates memory for the array

Do k=1,N
Do j=1,N
   Do i=1,N
      A(i,j,k) = ((i+j+k)/100.)       ! just an example
      summ =summ + A(i,j,k)**2        ! make sum of squares
   EndDo
EndDo
EndDo

write (*,*)  ' rms =',sqrt(summ/N**3)
summ = SUM(A**2)                      ! use a build-in function
write (*,*)  ' rms2=',sqrt(summ/N**3)
stop
end
```

```
/* ---------------------------------------------------------------
                              test memory: C- style
   ---------------------------------------------------------------- */
#include <stdio.h>
#include <math.h>
# define N 50
int main()
{    int i,j,k;
     float A[N][N][N];
     double sum=0;

     printf(" Required Memory=%12.6f%3s\n",4.*pow(N,3)/pow(1024.,2),"Mb");
     for(i = 0; i < N; i++)   {
     for(j = 0; j < N; j++)   {
        for(k = 0; k < N; k++){
          A[i][j][k] = ((i+j+k+3.)/100.);   /* just an example */
          sum +=  pow(A[i][j][k],2);     /* make sum of squares */
        }
     }
     }
     sum =sqrt(sum/pow(N,3))    ;
        printf(" rms=%14.9F\n",sum );
    return 0;
}
```

Example: Main program

Declarations are in file nbody1.h

Open files

Set parameters

Read data

Analyze and print

```
c-------------------------------------------------------------------
c                    Set Initial Conditions for N-body simulations
c                    Uses Jeans equation for spherically symmetric
c                    configuration with isotropic velocities
c
c-------------------------------------------------------------------
      INCLUDE 'nbody1.h'
      REAL*8    Ekin,Epot,Etot,yc(3)

                              ! open files
      Open(1,file='DATA/particles.dat',form='unformatted')
      Open(3,file='DATA/energies.dat')
      Open(8,file='DATA/analysis.dat')
      Open(11,file='DATA/initial.dat')

                              ! set parameters
      Np          = 10000
      time        = 0.
      dt          = 3.d-3
      epsilon     = 2.d-3
      RunTime     = 10.
      eps2        = epsilon**2
      do i=1,3          ! offset for the center of snapshot
         yc(i)    = 0.
      EndDo
            write(3,30)             ! write headers for files
            write(8,40)

            rs =0.25
            Rmaxinit =10.
            Rmininit =0.001
            aMassTotal=1.
      Call Initial(aMassTotal)
      Call Energies(Ekin,Epot)

            Etot = Ekin+Epot
            write (*,*)  ' Etot=',Etot
            write(3,20) time,iStep,Ekin,Epot,Etot,Ekin/Etot

      indx =0
      Call Analyze(indx,yc)
      Call WriteSnapshot(yc)
      Call SaveMoment

      write(*,'(80("-"))')
      write (*,*)'-----------   INITIAL CONDITIONS      -------------'
      write(*,*) ' Number of particles=',Np,' Time=',time
      write(*,*) ' Time step          =',dt,' Step=',iStep
      write(*,*) ' Force softening    =',epsilon
```

C: every statement must be terminated by ';'

A statement can span many lines. There is no need to indicate that the statement continues on the next line

```
  printf("Hello world"); return 0;
```

is the same as

```
  printf("Hello world");
  return 0;
```

which is the same as

```
  printf (
  "Hello world") ;


  return 0;
```

Fortran: A statement can span few lines.

To indicate that the statement continues on the next line put '&". Few statement can be placed on the same line. In this case they must terminated by ';"

```
  write(*,'(a)') ' Hello world'
```

which is the same as

```
  write (*,'(a)')  &
                'Hello world'
```

```
  x(i) =  25.  ; y(i) =i ; z(i) = sin(pi*3.)
```

Comments

C:

C has two types of comments : 1. **Multi-line comment** : A comment may begin with /* and end with */. This type of comment may span part of a line or may span many lines. 2. **Single line comment** : For a one line comment statement, you can use the token "//" at the beginning of the statement. The compiler ignores all text from a "//" until the end of a line.

Fortran:

No multiple line comments. Token “!” means everything after the token is a comment

Writing comments

- Rule Number One: write text in such a way that *just looking* at it one sees the structure of the code. No comments are needed to understand what is written.
- Rule Number Two: follow rule N1, but then write short comments: Always. Write them as you type. You will not come back to add a comment.
- Rule Number Three: comment every subroutine or function by putting dividing lines, which clearly separate the text.
- Rule Number Four: no dividing lines in the main text. Comments should not obstruct the text
-

C: C distinguishes cases. For example variables My and my are different

Fortran: No respect to cases: My, MY, and my are all the same

There are local and global variables and arrays.

C: Global

These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

DEFINING GLOBAL VARIABLES

```
/* Demonstrating Global variables  */
#include <stdio.h>
int add_numbers( void );              /* ANSI function prototype */
/* These are global variables and can be accessed by functions from this point on */
int  value1, value2, value3;
int add_numbers( void )
{
        auto int result;
        result = value1 + value2 + value3;
        return result;
}
main()
{
        auto int result;
        value1 = 10;
        value2 = 20;
        value3 = 30;
        result = add_numbers();
        printf("The sum of %d + %d + %d is %d\n",
                value1, value2, value3, final_result);
}
```

Sample Program Output

The sum of 10 + 20 + 30 is 60

There are local and global variables and arrays.

C: Global

The scope of global variables can be restricted by carefully placing the declaration. They are visible from the declaration until the end of the current source file.

```
#include <stdio.h>
void no_access( void ); /* ANSI function prototype */
void all_access( void );

static int n2;              /* n2 is known from this point onwards */

void no_access( void )
{
        n1 = 10;        /* illegal, n1 not yet known */
        n2 = 5;         /* valid */
}

static int n1;              /* n1 is known from this point onwards */

void all_access( void )
{
        n1 = 10;        /* valid */
        n2 = 3;         /* valid */
}
```

Declarations

Fortran: Global parameters and arrays are handled by 'modules'. A module may contain, among other things, a list of variables, which are available to every program unit, which uses the module.

Global variables are placed in 'heap' or DATA segment of memory. By default, local variables are placed in STACK. Memory available for STACK can be small. Adding attribute SAVE to declaration of a local array moves the array to 'heap' and also makes the array re-usable.

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5)
END MODULE MOD_A
...
SUBROUTINE SUB_Z
  USE MOD_A                  ! Makes scalar variables B and C, and array
  ...                        !   E available to this subroutine
END SUBROUTINE SUB_Z
```

AUTOMATIC AND STATIC VARIABLES

C programs have a number of segments (or areas) where data is located. These segments are typically,

_DATA Static data
_BSS Uninitialized static data, zeroed out before call to main()
_STACK Automatic data, resides on stack frame, thus local to functions
_CONST Constant data, using the ANSI C keyword const

The use of the appropriate keyword allows correct placement of the variable onto the desired data segment.

```
/* example program illustrates difference between static and automatic variables */

#include <stdio.h>

void demo( void );              /* ANSI function prototypes */

void demo( void )
{
        auto int avar = 0;

        static int svar = 0;

        printf("auto = %d, static = %d\n", avar, svar);

        ++avar;

        ++svar;
}
```

Static variables are created and initialized once, on the first call to the function. Subsequent calls to the function do not recreate or re-initialize the static variable. When the function terminates, the variable still exists on the _DATA segment, but cannot be accessed by outside functions.

Automatic variables are the opposite. They are created and re-initialized on each entry to the function. They disappear (are de-allocated) when the function terminates. They are created on the _STACK segment.

Loops

Programming with C and Fortran

C:

```c
int a=1;
while(a<100) {
   printf("a is %d \n",a);
   a = a*2;
}
```

```c
for(ix = 1; ix <= 10; ix++) {
   printf("%d ", ix);
}
```

Fortran:

```
DO WHILE (I > J)
  ARRAY(I,J) = 1.0
  I = I - 1
END DO
```

```
Do ix = 1,10
   print *, ix
End Do
```

```
DO I =1, 10
  A(I) = C + D(I)
  IF (D(I) < 0) CYCLE   ! If true, the next statement is omitted
  A(I) = 0              ! from the loop and the loop is tested again.
END DO
```

```
LOOP_A : DO I = 1, 15
  N = N + 1
  IF (N > I) EXIT LOOP_A
END DO LOOP_A
```

C:

Fortran:

Relational and Equivalence Expressions:

a < b
if **a** is less than **b**, 0 otherwise.
a > b
if **a** is greater than **b**, 0 otherwise.
a <= b
if **a** is less than or equal to **b**, 0 otherwise.
a >= b
if **a** is greater than or equal to **b**, 0 otherwise.
a == b
if **a** is equal to **b**, 0 otherwise.
a != b
if **a** is not equal to **b**, 0 otherwise

C:

```
if(a > b) {
   c = a;
} else if(b > a) {
   c = b;
} else {
   c = 0;
}
```

Fortran:

```
IF (expr) THEN
      block
[ELSE IF (expr) THEN
      block]...
[ELSE
      block]
END IF
```

C:

```c
switch(/* integer or enum goes here */) {
  case /* potential value */:
    /* code */
  case /* a different potential value */:
    /* different code */
  /* insert additional cases as needed */
  default:
    /* more code */
}
```

```c
switch ( someVariable ) {
case 1:
  printf("This code handles case 1\n");
  break;
case 2:
  printf("This prints when someVariable is 2, along with...\n");
  /* FALL THROUGH */
case 3:
  printf("This prints when someVariable is either 2 or 3.\n" );
  break;
}
```

Fortran:

CASE Constructs

The CASE construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a SELECT CASE statement. The CASE construct takes the following form:

SELECT CASE (*expr*)
[CASE (*case-value* [, *case-value*]...)
    block]...
[CASE DEFAULT
    block]
END SELECT

```
INTEGER FUNCTION STATUS_CODE (I)
   INTEGER I
   CHECK_STATUS: SELECT CASE (I)
   CASE (:-1)
     STATUS_CODE = -1
   CASE (0)
     STATUS_CODE = 0
   CASE (1:)
     STATUS_CODE = 1
   END SELECT CHECK_STATUS
END FUNCTION STATUS_CODE
```

Archaic Features, which should be avoided

- Labels or line numbers should be avoided. Code can jump to any label (goto N). Because compiler does not know from where you can jump in to the labeled statement, it does not do many optimizations. This makes code very slow.
- Compilers try to optimize pieces of the code, which do not contain labels.
- Do not abuse the rule: sometimes it is easier to use labels.

Implicit statement

- It is often recommended (in strong words) not to use "Implicit" statement. The main motivation is that one can make a typo in a name of a variable, which is difficult to find. 'Implicit none' is recommended for the beginners.
- This recommendation comes from people, who do not write large and complicated codes.
- 'Implicit' is a powerful tool, which makes coding more efficient. Yet, one should always use it or never use it. Do not mix explicit declarations and implicit style.
- Always follow implicit rules: variables and arrays starting with letters (i-n) are INTEGER. Do not define Mass as real. 'aMass' is as clear as 'Mass'.

Formal parameters and common blocks

- Data can be passed to a subroutine either using formal arguments (e.g. *CALL MySUB(x,y,z)*) or by as global variable(e.g. *COMMON /A/x,y,z)* or in module)
- Those are very different mechanisms. Formal arguments are handled through ‘stack’ and common blocks (also SAVE and ALLOCATE) go to the ‘heap’. Most of the computer RAM is allocated to the ‘heap’ because by design stack is for small things.
- Use formal arguments for small arrays, few arguments, and for constructs, which cannot be handled by common blocks.

http://www.math.utah.edu/software/c-with-fortran.html#common-types

Argument addressing In Fortran, **all routine arguments, without exception, are passed *by address***. This means that within a FUNCTION or SUBROUTINE, assignment to an argument variable will change its value in the calling program. The exact behavior depends on whether arguments are handled by direct reference, or by copying into local variables on routine entry, and copying back on exit. Code like this

```
CALL FOO(3)
III = 3
PRINT *,III
...
SUBROUTINE FOO(L)
L = 50
END
```

will print 3 on some systems, and 50 on others, with surprising effects in the rest of the program from instances of 3 in the source code having been silently changed to 50.

Fortran offers no way to avoid such problems: arguments passed to other routines are always subject to modification in the called routines!

In C and C++, **scalar objects are passed *by value*, and array objects *by address* of the first (index zero) element**.

In C, C++, and Fortran, arguments that are themselves routine names are passed *by address*. If we rewrite the above Fortran sample in C

```
   foo(3);
   iii = 3;
   printf("%d\n", iii);
...
void foo(int l)
{
    l = 50;
}
```

the program will always print 3, since function foo(l) has no access to the original location of its argument. The change to the argument is entirely local to the function foo(l).

Language memory management

Most programs in C and C++ make heavy use of dynamically allocated memory, in C through the **malloc() / free()** family, and in C++ through the **new** operator (the C memory allocators are available in C++, but strongly deprecated). However, *neither language garbage collects memory that is allocated, but no longer in use. Such memory leaks plague most long-running programs written in these languages*.

Programs in Fortran use a combination of declaration ALLOCATABLE/ALLOCATE (and then DEALLOCATE) to manage dynamical memory. Memory leaks can occur if array is not deallocated before leaving programming unit that allocated the array.

http://www.math.utah.edu/software/c-with-fortran.html#common-types

Fortran and C/C++ common data types

For mixed-language programming in C/C++ and Fortran, only the minimal intersection of their many data types can be relied on:

- int == INTEGER == INTEGER*4,
- float == REAL == Real*4,
- double == DOUBLE PRECISION == Real*8

Array indexing differences

Fortran array indexing starts at one, while C/C++ indexing starts at zero. An N-by-N matrix is declared in Fortran as typename A(N,N), and indexed from 1 to N, while a corresponding C/C++ array is declared as typename a[N][N], but indexed from 0 to N-1. This feature of C/C++ is a frequent source of off-by-one errors, and when you mix C/C++ code with Fortran, you are even more likely to be confused.

Function return types

Fortran has SUBROUTINE, which does not return a value, and must be invoked by a CALL statement, and FUNCTION, which returns a scalar value, and must be invoked by function-like notation in an expression. While some Fortran implementations permit a FUNCTION to be referenced by a CALL statement, it is decidedly nonportable to do so, and is a violation of all national and international standards for Fortran.

Fortran functions can return only scalar values, not arrays.

C and C++ have only functions, but those `functions' may or may not return a value. Good programming practice declares non-value-returning functions to be of type void, and value-returning ones to be of any non- void scalar type.

Array storage-order differences

Fortran stores arrays in row order, with the first subscript increasing most rapidly. C and C++ store arrays in column order with the last subscript increasing most rapidly. Despite arguments to the contrary, there is really no reason to favor either storage convention: rows and columns are equally important concepts.

http://www.math.utah.edu/software/c-with-fortran.html#common-types

The Fortran view of files

Fortran files are of two fundamental types: **FORMATTED** (text) and **UNFORMATTED** (binary).

Binary files are compact, fast to read and write, and incomprehensible to humans. They also avoid data conversion and accuracy loss, since data is stored in such files with exactly the same bit patterns as in memory.

Text file properties are the opposite of these, but text files have the advantage of being highly portable, and readable (and editable) by humans.

Data in Fortran files is divided into ***records***, which are recognizable objects in the language and run-time libraries, and are logically the smallest objects in files that the language recognizes.

For text files, line boundaries mark records, and such files are generally trivial to process with any programming language or text software tool.

For Fortran binary files, special markers (usually 4 to 12 bytes in length) must be recorded in the files to mark the start and end of records, where a `record' corresponds to the data that is in the I/O list of a Fortran READ or WRITE statement.

A Fortran binary READ (unit) statement with no I/O list can be used to skip forward over records, and a BACKSPACE statement may be used to skip backward over records.

Both text and binary file types may be accessed sequentially or randomly, although with random access, the records must be of uniform length, and in older Fortran implementations, and even on some current operating systems, the number of records must be known and declared at file-open time.

Standard Fortran does not define any way to read or write data at the byte level.

The C/C++ view of files

In C and C++ a file is viewed as an ***unstructured stream of zero or more bytes***, and the C fgetc() and fputc() functions can access a byte at a time. Any additional required file structure must be supplied by the user's program itself.

This low-level view of files as unstructured byte streams makes it simple in C and C++ to write files that have any desired structure, since the user has complete control over exactly what bytes are read or written. Nothing is added or subtracted by the run-time library, with the sole exception of text files on systems that do not use a simple ASCII LF character to delimit lines. On such systems, the C \n (newline) character may be mapped to something else on output (CR on Apple Macintosh, CR LF on IBM PC DOS, Microsoft Windows, and several historic now-retired operating systems, or possibly a length field recorded at the start of the record (common practice in VAX VMS and old PDP-11 text files)). However, even on those systems, the file can be opened in binary mode, suppressing all such translations, and giving the user complete control over the data stream.

Formatting I/O

- Fortran has extensive tools and options to read and write files. There is nothing like that in C.

Format can be given as a separate statement or as an in-line specification

```
write (*,10) a,b,c
10 format(f8.2,f9.2,f10.5)

write (*,'(f8.2,f9.2,f10.5)') a,b,c
```

Implicit Do loops in formats

- You can write in a file n=1 an array *Z* of length *N* in two ways:

  write(5) Z or write(5) (Z(i),i=1,N)

  The first form is faster, but the second is more flexible if we want to modify it:

  write(5) (Z(i),i=1,N-5,3)

- More complicated example:

  write(5) a,b,c,((Z(i,j),i=1,N,10),j=2,M/2)

Formatting outputs

- To produce nice-looking output tables it is convenient to use ‘T’ format. For example,

write(,’(f8.2,T30,a,g12.3)’) r,’Density=’,d*

starts text ‘Density’ at the position 30. This is useful for wide tables with many headers

Files

- Formatted and Unformatted
- Sequential and direct access
- position='append'

```
Open(20,file='myfile.dat')
```

Unit 20 is attached to sequential, formatted file myfile.dat

```
Open(20,file='myfile.dat',   &
      form='unformatted)
```

Unit 20 is attached to sequential, unformatted file myfile.dat

```
write(20,*) x,y,z
```

Write x,y,z in to Unit 20. Because it is attached to file.dat, the write command modifies myfile.dat

```
read(20,*,error=10,end=10) x,y,z
```

Read from Unit 20. If error of end of file happen, go to statement labeled 10

```
Close(20)
```

Compilation

Compile and link: `ifort -O2 file1.f file2.f -o file.exe`

Compile:
```
ifort -O2 -c file1.f
ifort -O2 -c file2.f
```

link: `ifort -O2 file1.o file2.o -o file.exe`

Compile for parallel execution with OpenMP:

`ifort -O2 -openmp -parallel file1.f file2.f -o file.exe`

Parallel execution with OpenMP:

```
setenv OMP_NUM_THREADS 16
file.exe
```

```
export OMP_NUM_THREADS=16
file.exe
```