PSTI RESEARCH LECTURE SERIES

"SCIENTIFIC COMPUTING WITH FORTRAN 95"

DR. VIKTOR K. DECYK

RESEARCH PHYSICIST, UCLA

Abstract

The Plasma Science and Technology Institute (PSTI) at UCLA will host a Research Lecture Series during the Spring Quarter of 2002 highlighting the expertise of one of its distinguished senior researchers, Dr. V.K. Decyk. The weekly lectures are open to all interested persons. Students who wish to receive instruction credit can enroll in the course "Physics 290" for S/U grade.

"Fortran 95" is a powerful, high level language designed for mathematical modeling. It is a safe language that gives high performance, with many features attractive to scientists who want to spend a minimum amount of time programming, such as leak-proof dynamic memory. Although it is not a fully object-oriented language, it supports enough features that object-oriented design can be implemented. This is important because it allows more complex software to be constructed. These lectures are intended to introduce "Fortran 77" programmers to the new features of " Fortran 95". Strategies for how to modernize old "Fortran 77" codes incrementally, as well as object-oriented design in "Fortran 95" will be discussed.

First meeting: April 3, 2002

Schedule: Every Wednesday, from 3-4:30 p.m.

Location: Kinsey 141

Contacts: Decyk@physics.ucla.edu, http://www.physics.ucla.edu/psti

Course outline:

First third:

Overview of new features in Fortran90/95.

Second Third:

Overview of Object-Oriented Concepts

Final Third:

Strategies for incremental application of OO concepts to Fortran90 programs, development of frameworks

Recommended books:

T. M. R. Ellis, Ivor R. Philips, and Thomas M. Lahey, **Fortran90 Programming** [Addison-Wesley, 1994].

Michael Metcalf and John Reid, **Fortran90/95 Explained** [Oxford Univ. Press,1996].

T. M. R. Ellis, Ivor R. Philips, **Programming in F** [Addison-Wesley, 1998].

The first part of the course will be largely based on the F language. This is a subset of Fortran90, which contains most of the new good stuff and omits the older deprecated stuff. It is relatively simple, yet contains powerful modern constructs. This simplicity comes in part from requiring only one way to do things, where Fortran90 might allow multiple ways. Because of its simplicity, compilers are inexpensive and they have better error reporting. All valid F programs are also valid Fortran90 programs.

Information about F compilers can be found at:

http://www.fortran.com/fortran/imagine1/

There are only minor differences between Fortran95 and Fortran90

Why Fortran?

- Fortran95 is not your father's Fortran

- Designed for doing mathematics, not web browsing

- Simple language which gives high performance

- Safe language (less error prone), e.g., leak-proof dynamic memory

Physicists are part-time programmers, not writing commerical software, mainly interested in getting new result with minimum effort

Isn't Fortran dead?

- Vendors of Fortran report sales are not going down.

Some general new features about Fortran90

It is 100% backward compatible with Fortran77. Old programs should work without modification.

The correct spelling is now Fortran, not FORTRAN.

free form is now permitted,

& used as a continuation

exclamations delimit comments, which are allowed at the end of a line

```
a = 1.    ! this is a comment
```

; allows multiple statements on a line:

```
a = 1.; b = 2.
```

underscores can be used in names, my_sub is a valid name

Many new intrinsic functions are standard:

system_clock(), wall clock time
random_number(), uniform random number

Types (F book, chapter 3)

Five intrinsic types: integer, real, logical, complex, character. Numbers are stored differently internally when integer or real.

Fortran allows default types based on names. All types are implicitly real unless the name begins with the letters i,j,k,l,m,n, in which case they are integer (i-n are the first two letters in the word integer). However, it is a bad idea not to declare all variables, because it prevents the compiler from finding some of your mistakes.

```
implicit none       ! means all variables must be declared
integer :: i, j
real :: a
```

Type determines what kind of operands are allowed.

Division "/" means something different with integer, real, and complex. This is called operator **overloading**, or static **polymorphism**.

Implicit type conversion between intrinsic types can occur in expressions, e.g., i = j + 1. will actually convert the j to a real, add 1., then convert back to an integer. Such conversions can be expensive, e.g., i = j**2. will use logs to calculate the result (note the . after the 2), whereas i = j**2 will just multiply.

You can explicitly convert types using intrinsics such as real(), int(), or cmplx(), e.g., i = int(a) or a = real(i)

Double precision

```
double precision :: d       ! or sometimes, real*8 d
```

is still supported, but deprecated. Kind parameters are introduced to discriminate among kinds of reals, etc.

```
real(kind_num) :: d      ! or real(kind=kind_num) :: d
```

There are different kind_nums for different precisions, but the bad thing is that the actual value of kind_num is not defined, and different compilers can use different values. To determine what the value should be, you have to use the selected_real_kind intrinsic, usually in the declaration:

```
integer :: kind_num = selected_real_kind(10,60)
```

This will give the kind_num used by the compiler to give a real which has at least 10 digits of accuracy and an exponent of at least 60. This is likely to give you an 8 byte real. However, it might not. It could give you a 16 byte real, you have no way of knowing. All you know is that you get at least this much precision. If the requested precision is not available, you get a compiler error. All this is very confusing in my opinion. It is makes it hard to call other languages. However, in practice, most compilers use kind numbers which are byte lengths,

```
real(8)              ! instead of real*8.
```

The good news is that it is now possible to have different integer kinds (1 byte, 8 byte, etc.), and other types.

Character types

```
character(len=5) :: c
```

The only operator for characters is concatenation "//".

```
c = 'abc'//'de'       ! "abd"//"de" is also allowed.
```

gives c = 'abcde'.

Substrings are also allowed:

```
c(2:3)          ! selects 'bc'.
c(4:4)          ! selects 'd'
```

Assignments of different lengths result in truncation or padding.

```
c = 'abcdef'     ! gives 'abcde'
c = 'abcd'       ! gives 'abcd ', with a trailing blank
```

Character kinds are also possible, e.g.,

```
character(len=24,kind=kanji) :: japanese
```

but they may not necessarily be one byte per character

Since the size of characters is fixed at declaration, the trim instrinsic is useful to delete trailing blanks.

If c = ‘abcd ‘, (note trailing blank) then

trim(c)

gives ‘abcd’ without the trailing blank.

Adjustl (left justify) is a useful intrinsic for removing for leading blanks. If c = ‘ abc ‘, (note leading and trailing blank) then,

c = trim(**adjustl**(c))

gives ‘abc’ without leading or trailing blanks.

Program Units (F book, chapter 4)

```
      subroutine roots(a,b,state)
      implicit none                        ! declare all variables
! declare arguments being passed in or out
      real, intent(out) :: a               ! write only
      real, intent(in) :: b                ! read only
      integer, intent(inout) :: state  ! read and write possible
! declare local variables
      real :: scratch                      ! this will disappear on exit
      real, save :: last_state             ! this will stay on exit
      real, parameter :: pi = 3.14159  ! cannot be changed
      ...          ! contents of procedure go here
      end subroutine roots      ! return statement not needed
```

Local variables generally go on a stack, memory used by run time system to put subroutine argument addresses and local variables. When the subroutine returns the memory in the stack is released. If one subroutine calls another, the first stack space is kept, and more stack space is added for the second subroutine. Static variables might be kept in a heap, a pile of memory which is randomly used and released, as needed.

Interface blocks allow Fortran90 to check at compile time if arguments are valid when a subroutine is called.

```
program my_program

real :: x = 1.0, y = 2.0, z = 3.0
integer :: s = 1

interface
   subroutine roots(a,b,state)
  implicit none          ! only arguments go here
   real :: a, b
   integer :: state
   end subroutine
end interface
                         ! the arguments will be checked
call roots(x,y,z)        ! compiler will detect that z is wrong

end program
```

This functions like header files (.h files) in C. **Interface** refers to a procedure and its argument types

Interface blocks are primarily useful for calling old legacy routines or procedures for which you don't have the source code. For new functions, the use of modules makes type checking automatic and interface blocks unnecessary.

If you lie in the interface statement, the compiler will guarantee the procedure is called incorrectly!

Modules

A module is a new program unit for grouping together subroutines, data, etc. We will use this construct to build classes later.

```
module my_module
 real, parameter :: pi = 3.14159      ! data goes here
 contains                             ! procedures go here
 subroutine roots(a,b,state)
implicit none
 real, intent(out) :: a
real, intent(in) :: b
integer, intent(inout) :: state
 ...          ! contents of procedure go here
            ! Something new: pi is available here
 end subroutine roots
end module my_module

program my_program
use my_module  ! make information in module available
                        ! pi and roots are now known here
 real :: x = 1.0, y = 2.0, z = 3.0
 integer :: s = 1
                          ! the arguments will be checked
 call roots(x,y,z)      ! interface block no longer needed
                          ! z is still the wrong type.
 end program
```

Modules can be compiled separately. Modules are extremely useful and powerful.

Functions result variable no longer has to be the name of the function:

```
function cube_root(x) result(root)
real, intent(in) :: x
real :: root              ! result variable is called root
...               ! contents of function go here
end function cube_root   ! return statement not needed
```

Subroutines and functions can call themselves, if the recursive keyword is used.

```
recursive subroutine factorial(n,factorial_n)
integer, intent(in) :: n
integer, intent(out) :: factorial_n
 ...
call factorial(n-1,factorial_n)
factorial_n = n*factorial_n
 ...
end subroutine factorial
```

Recursive subroutines can be useful for rapid development of certain algorithms. However, they can have very high overhead, and their performance is often poor.

Flow Control (F book, chapter 5)

New logical symbols:

```
>      ! same as .gt.
<      ! same as .lt.
==     ! same as .eq.
/=     ! same as .ne.
```

If-then-else constructs are evaluated in order. The first true one is executed.

```
logical :: logical1, logical2
logical1 = (a>b).or.(b<c)

if (logical1) then
   ...
else if (logical2) then
   ...
else
   ...
endif
```

Case construct is similar but more restricted, requires mutually exclusive choices.

```
integer :: i
select case (i)
case (1)
   ...
case (2)
   ...
case default
   ...
end case
```

The case selector can only be an integer, logical, or character expression. Because the possible choices are known at compile time, it is possible for the compiler produce code to jump to the correct case directly without any tests. The if-then-else construct is more general, but slower.

Ranges are also allowed in case statements, e.g.,

```
case(-1:)      ! i less than zero
case(1:3)      ! i between 1 and 3.
case(:4)       ! i greater than 3
```

Iteration (F book, chapter 6)

```
do i = initial, final [, increment]
   a(i) = ...
enddo
```

numerical labels no longer required. Loop variable cannot be modified. If final < initial, loop is not executed.

Some useful character intrinsics include len(c) to obtain the length of c, char to convert an integer code to a character and ichar to convert a character to an integer code. For example to convert from upper case to lower case on an ascii machine, one can write:

```
do i = 1, len(c)
  c(i:i) = char(ichar(c(i:i) + 32))
enddo
```

More flexible loops are possible, with Exit and Cycle statements:

```
do i =1, max
   ...
   if (s>1.) exit        ! exit before i = max
   ...
end do
```

```
do i =1, max
   ...
   if (s>1.) cycle      ! go to next iteration
   ...                  ! this part skipped if cycle executed
end do
```

Infinite loops with an exit are possible

```
do
if (logical1) exit
...
end do
```

This is equivalent to a do while loop, which is also possible

Loops can be labeled:

```
do_name: do i= 1, n
   ...
      end do do_name
```

If-then-else can also be labeled in this way. This is useful for deeply nested do loops and if-then-else constructs.

Useful new intrinsics:

epsilon(x), smallest number < 1, but not zero.
huge(x), largest number possible of this type
tiny(x), smallest positive number of this type
digits(x), number of significant digits of real or integer
precision(x), decimal precision of real or complex