Some important extensions of Fortran-90 over Fortran-77

The goals in scientific programming are first to get correct answers and second to write code that is modular, clear, and can be modified later. These goals can be achieved in any of a variety of languages, and students may use any language for which we have a compiler on physics. While I shall give examples in the C programming language, some students may prefer to use C++ or Fortran-90.¹

Some older variants of Fortran, such as Fortran-IV and Fortran-77, were popular in scientific programming before 1980. Indeed, we’ll make use of some of numerical libraries, such as LAPACK, that have been implemented in Fortran-77. While parts of the projects can be successfully completed in Fortran-77, the language lacks several essential capabilities. Students who decide to use Fortran-77 will need either to write parts of their code in a language like C that has these features and then link the object files together² or else use Fortran-90. Fortran-90 claims all of Fortran-77 as a (small) subset, so it should not be too painful to upgrade. To invoke a Fortran-90 compiler on physics, use the command pgf90. The most common arguments are the same as for the Fortran-77 compiler, f77, or the C compiler, cc.

Fortran-90 is a very large language, and I cannot hope even to summarize its extensions over Fortran-77. However, there are three features that will be enormously useful in this class.

1. In all operating systems used in scientific research (mostly Unix, but also VMS and DOS), the primary means by which the program receives input from the user or from other programs is through the command line. For example, to tell myprog that it should use a $23 \times 23$ matrix, I might invoke it as

   prompt% myprog 23

   Amazingly, no version of the Fortran standard contains any provision for reading the number 23 from command line. There appears to be a de-facto standard using function IARGC and subroutine GETARG, illustrated below (Figure 1). It worked on two different compilers that I tried, but there’s no guarantee that it will work on all compilers. Figure 2 shows how to convert a numerical command-line argument (such as “23”) from a string to a number.

2. Most scientific programs have arrays (one or two-dimensional) of a size that depends on some parameter in the problem at hand, such as the number of sites in a simulation. This parameter is variable and might not be known until the program is running. Old-fashioned Fortran-77 programmers were forced either to recompile their programs for each new set of parameters (which is time consuming) or to fix the maximum dimension each array is ever “likely” to achieve. Either technique is prone to error. The ALLOCATE statement in Fortran-90 lets the program allocate memory as it is needed. Figure 2 illustrates dynamic memory allocation.

¹ Fortran-90 and C++ are examples of “object-oriented” languages, which provide certain advantages over C and Fortran-77, such as operator overloading. They also have enormously more syntaxic elements. In my experience as a computer-center manager, I found that each programmer writing in C++ knew a different subset of the language, and that computer-science students who had learned only C++ tended to misuse language features, resulting in subroutines that went on for pages, could not be read or modified by anyone else, and were in no sense modular. This is what I call “object-disoriented programming” and is part of the reason I do not recommend C++ or Fortran-90 for new programmers.

² On physics, give the command “help linking languages” for information on linking C and Fortran.
Demonstrate command-line reading in Fortran.
This program echoes each command-line argument on a separate line.
usage: myprog [arguments]
!
PROGRAM MAIN
IMPLICIT NONE
CHARACTER(15) :: BUF ! truncate at 15 characters
INTEGER :: I, J, IARGC
I = IARGC() ! get the number of arguments
DO J=1,I
    CALL GETARG(J, BUF) ! get the j'th argument,
    WRITE(6, *) BUF ! and write it out
END DO
END

3. Old versions of Fortran provided a limited number of scalar data types: INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER. Sometimes, it is useful to package different types together; it is almost essential in order to implement data structures such as linked lists. As a simple example, one might have a database of employee records, where each employee had a name, a salary, and the year of hire. One would like a single variable type with three fields: CHARACTER for the name, REAL for the salary, and INTEGER for the year of hire. The old COMMON-block mechanism was woefully inadequate, since only one package of each “type” could exist. The TYPE keyword of Fortran-90 (analogous to struct in C) allows the user to create a new data type. Figure 3 shows a Fortran-90 code that does the same thing as the bintree.c example in the Week-1 packet. Bintree.f90 also demonstrates a skeletal implementation of variable-length strings.

In writing in Fortran (whether Fortran-90 or Fortran-77), the programmer should avoid bad habits common to archaic Fortran-IV/66 codes. These include lack of comments (use the ! feature, which allows comments on the same line as code), subroutines that go on forever (try to limit each subroutine to 24 lines, so it will fit in one standard xterm window), and the use of magic numbers. Above all, declare every variable explicitly, and begin every procedure with the line

IMPLICIT NONE

(See the lower-left box on page 3 of the week-1 packet.)
The examples in this brochure are available on physics in the directory
/home/5156/examples/fortran90
Demonstrate memory allocation in Fortran-90.

Allocate, fill, and print out a one-dimensional array of arbitrary length specified on the command line.

usage: allocdemo n

where n is the length

PROGRAM MAIN
IMPLICIT NONE
CHARACTER(15) :: BUF
INTEGER, POINTER, DIMENSION(:) :: A ! pointer to 1-dim array
INTEGER :: N, IARGC ! N=the length
IF(IARGC().LT.1) GOTO 1 ! no argument: error
CALL GETARG(1,BUF) ! get the argument
READ(BUF,*) N ! convert string->number
IF(N.LT.1) THEN ! N must be >= 1
1 STOP 'SYNTAX: ALLOCDEMO N' ! die
ENDIF
ALLOCATE(A(N)) ! allocate N integers
CALL DOFILL(A,N) ! fill it
CALL DOPRINT(A,N) ! print it out
DEALLOCATE(A) ! done with it
END

! Fill an array with integers from 1 to its length
SUBROUTINE DOFILL(P,N)
IMPLICIT NONE
INTEGER :: N, P(N), I
DO I=1,N
   P(I) = I ! fill it
END DO
END

! Print out an array of length N
SUBROUTINE DOPRINT(P,N)
IMPLICIT NONE
INTEGER :: N, P(N), I
DO I=1,N
   WRITE(6,*) P(I)
END DO
END

Figure 2. Example for numbered paragraph 2.
contains
module node_module
! Find the correct place to insert a new node, and do so.
! root is a pointer to the root node (or nil)
! call placenode(s, root)

IMPLICIT NONE
character(+), intent(in) :: s
type(node), pointer :: root
if(associated(root)) then ! root already present?
call placenode(s, root) ! hard case
else
call placenode(s, root) ! easy case
end if
end subroutine placenode

placenode() calls this when root points to nothing
placenode(s, root)

IMPLICIT NONE
character(+), intent(in) :: s
type(node), pointer :: root
allocate(root) ! allocate a node
call assemode(root, s) ! fill it
end subroutine placenode

placenode() calls this when root exists — call sequence as placenode's

Three hacks: (1) instead of writing comparison interfaces in the
! string module between strings and character(+)'s, I convert
! the string back to a character(+)'s. (2) Pointers to pointers
! are really mess in Fortran-90, so I use 'flag' to keep track
! of whether the most recent traversal was to the left or to
! the right. (3) Strings longer than 256 won't be compared correctly
placenode(s, root)

IMPLICIT NONE
character(+), intent(in) :: s
type(node), pointer :: root
if(associated(root)) then ! root already present?
call placenode(s, root) ! hard case
else
call placenode(s, root) ! easy case
end if
end subroutine placenode

! placenode() calls this when root points to nothing
placenode(s, root)

IMPLICIT NONE
character(+), intent(in) :: s
type(node), pointer :: root
allocate(root) ! allocate a node
call assemode(root, s) ! fill it
end subroutine placenode