Focus Manuals: Programing Guide: Features of Fortran 90 used in BIEF

links from Programing Guide

We briefly explain hereafter features of Fortran90 that are used in BIEF. For more detailed explanations please refer to a proper Fortran 90 book, such as ref.[5].

Structures

Fortran 77 only recognises integers, real numbers, boolean and character strings. Fortran 90 can be used to create structures. The following is an example of the creation of a **point** type structure composed of two real numbers, and a **circle** structure, composed of a **centre** and a **radius**:

```
TYPE point
   REAL :: x,y
END TYPE
TYPE circle
   TYPE(point) :: centre
   REAL :: radius
END TYPE
```

It can be observed that the **centre** is itself a structure of a type previously defined. Once the structure has been defined, objects of this type can be declared:

```
  TYPE(circle) :: ROND
```

ROND will be a **circle** with its **centre** and **radius**; the latter are obtained thanks to the %“component selector”. Thus the radius of ROND will be the real **ROND%radius**.

Pointers

Pointers are well known in C language, but are notably different in Fortran 90. Pointers in Fortran 90 may be used as pointers as in C but also as aliases. Unlike C, they are not mere addresses pointing to somewhere in the computer memory. The target must be defined precisely, for example the line:

```
  REAL, POINTER, DIMENSION(:) :: X
```

will define a pointer to a one-dimensional real array, and it will be impossible to have it pointing to an integer nor even to a 2-dimensional array. This pointer **X** will have then to be pointed to a target by the statement:

```
  X => Y
```

were **Y** is an already existing one-dimensional real array. Then **X** can be used as if it were **Y**, it is thus an alias.

X can be also directly allocated as a normal array by the statement:

```
ALLOCATE(X(100))
```

to have (for example) an array of 100 values. In this case **X** and its target have the same name.

A well known problem in Fortran 90 is the fact that arrays of pointers do not exist. To overcome this problem, one has to create a new structure which is itself a pointer, and to declare an array of this new structure. This is done for blocks, which are lists of pointers to **BIEF_OBJ** structures.

Modules

Modules are like **INCLUDE** statements, but are more clever, so that **INCLUDE** should now always be avoided. As a matter of fact, modules can be used to define global variables that will be accessible to all routines. With the following module:

```
MODULE EXAMPLE
   INTEGER EX1,EX2,EX3,EX4
END MODULE EXAMPLE
```

all the subroutines beginning with the statement **USE EXAMPLE** will have access to the same numbers **EX1**, ... **EX4**. With **INCLUDE** statements, it would be only local variables without link to **EX1** ,... declared in other subroutines.

Modules will thus be used to define global variables that will be accessed via a **USE** statement. If only one or several objects must be accessed, the **ONLY** statement may be used, as in the example below:

```
USE EXAMPLE, ONLY : EX1,EX2
```

This will enable to avoid name conflicts and secures programming.

Modules are also used to store interfaces that will be shared between several subroutines (see paragraph below).

Interfaces

Interfaces are a mean given to the compiler to check arguments of subroutines even if it has no access to them. For example, the following interface:

```
INTERFACE
   LOGICAL FUNCTION EOF(LUNIT)
      INTEGER, INTENT(IN) :: LUNIT
   END FUNCTION
END INTERFACE
```

says that function **EOF** has one integer argument. **INTENT(IN)** indicates that argument **LUNIT** is not changed. Interfaces of all BIEF subroutines have been put in a single module called **BIEF**. A **USE BIEF** statement at the beginning of a subroutine will prompt the compiler to check the arguments and also do some optimisations in view of the **INTENT** information (which can be **IN**, **OUT**, or **INOUT** depending on the use of the argument). If a function is declared in an interface, it must not be declared as an **EXTERNAL FUNCTION**.

Interface operator

New operations on structures could also be defined with the **INTERFACE OPERATOR** statement. For example a sum of two vectors as stored in BIEF could be defined so that the line:

```
CALL OS('X=Y ',U,V,V,0.D0)
```

could be replaced by:

```
U=V
```

Such interface operators have not been done in version 6.0, because operations like **U = A+B+C** would probably not be optimised and would trigger a number of unnecessary copies.

Optional parameters

Subroutines may now have optional parameters. Thanks to this new feature, subroutines **OS** and **OSD** of previous releases have been grouped in a single one. Hereafter is given the interface of new subroutine **OS**:

```
INTERFACE
   SUBROUTINE OS( OP, X , Y , Z , C , IOPT , INFINI , ZERO )
      USE BIEF_DEF
      INTEGER, INTENT(IN), OPTIONAL :: IOPT
      DOUBLE PRECISION, INTENT(IN), OPTIONAL, INFINI, ZERO
      TYPE(BIEF_OBJ), INTENT(INOUT), OPTIONAL :: X
      TYPE(BIEF_OBJ), INTENT(IN), OPTIONAL :: Y,Z
      DOUBLE PRECISION, INTENT(IN), OPTIONAL :: C
      CHARACTER(LEN=8), INTENT(IN) :: OP
   END SUBROUTINE
END INTERFACE
```

Subroutine **OS** performs on structure **X** the operation given in **OP**, e.g.

```
CALL OS('X=0 ',X=TRA01)
```

or:

```
CALL OS('X=Y ',X=TAB1,Y=TAB2)
```

Parameters **Y**,**Z** and **C** are used only for specific operations and otherwise are not necessary. When a parameter is missing and to avoid ambiguity, the parameters must be named, hence the X=TRA01 in the example above.

Parameters **IOPT**, **INFINI** and **ZERO** stem from the old subroutine **OSD** and are used only when a division is implied in the operation asked, for example if """OP = 'X=Y/Z '""". These 3 parameters are now optional. When they are present, it is better to name them as is done in the following line:

```
CALL OS('X=Y/Z ',U,V,W,0.D0,IOPT=2,INFINI=1.D0,ZERO=1.D-10)
```

The use of optional parameters will enable a better compatibility between different versions because

it will be possible to add a new parameter as an optional one.

Optional arguments will be written between brackets [] in argument lists in the rest of the document.