



FORTRAN 90

Andrew Longmore
TESSELLA SUPPORT SERVICES PLC

Issue V1.R1.M1
January 1997



FORTRAN 90

Introduction

In many quarters, FORTRAN is synonymous with the unstructured, primitive, or just plain ugly. You can't assign pointers, you can't allocate arrays dynamically, using common blocks is not the safest way of passing data around, and you have to leave that silly little gap at the start of every line.

No more. FORTRAN 90 is not just FORTRAN 77 with a few extensions, but a radical improvement on the language. It is still backward-compatible with F77, but only just. New language features include:

- 'Free source form' - i.e. free to start your code in column 1
- Derived types
- Modules (with public and private variables)
- Operator and assignment overloading
- Pointers
- Dynamic memory allocation
- Whole array operations.

Various things are still included in order to maintain backwards compatibility, but are obsolescent (i.e. may be removed in future versions) or redundant.

Derived Types

These are known as structures in other languages. For instance, suppose you wanted to store a point in space; it may be convenient to refer to that point as a single entity, rather than always as three co-ordinates. For example:

```
type:: point           ! derive the variable type 'point'  
  real :: x,y,z  
end type point
```

```
type (point) :: p1      ! define p1 to be a type 'point'  
type (point) :: origin = &    ! define origin as a  
  point(0.0, 0.0, 0.0)      !constant 'point'
```

```
p1%x = 1.0; p1%y = 2.0; p1%z = 3.0
```

Note that comments are shown by '!', the continuation across lines shown by '&', and the multiple statements on the same line shown by ';' (all part of the free source form). F77 users may be surprised by the '::' - these are optional for

internal types like ‘real’, but required for derived types.

Modules

One of the problems that could occur in F77 was if you used common blocks to make data accessible to two different parts of a program. The common block had to be defined in both places - if you changed the order of the variables, missed one out, or combined different data types, you’d be in trouble.

To get around this, many F77 programmers would define their common block in a separate file, and then include it wherever they wanted it. This would ensure it would always be the same, and if they wanted to change it they would only have to change it in one place. This is a close analogy to a module.

For instance, suppose we wanted to create our derived type ‘point’, maybe some other related derived types, and a whole load of procedures which go with them. We can put them all into a module called mpoint. To use that module, a procedure would then just have to have:

```
use mpoint
```

before its own declarations. It would then have access to all the objects in that module, unless the module had declared them as private or unless the procedure using the module only wanted some of them:

```
use mpoint, only : point
```

If you wanted to, another module, e.g. mshape, could use mpoint. The shape module could then have all the definitions and procedures to do with points, without ever knowing the internal workings of the mpoint module.

Operator and Assignment Overloading

Derived types are only part of the story. Once a type has been defined you can overload operators and assignment, and define new operators. This can be done in the modules. For instance, it might be useful if we could say:

```
use mpoint
type (point):: p1
p1 = 1.0
```

instead of:

```
use mpoint
type (point):: p1
p1 = point(1.0, 1.0, 1.0)
```

The module would then look something like this:

```
module mpoint
  implicit none
  ...
  type:: point
    real :: x,y,z
  end type point
  ...
  interface assignment (=)
    module procedure point_eq_r
  end interface
  ...
contains
  subroutine point_eq_r(p1,r1)
    real, intent(in) :: r1
    type(point), intent(out) :: p1
    p1%x=r1; p1%y=r1; p1%z=r1
    return
  end subroutine point_eq_r
  ...
end module mpoint
```

You can have more than one assignment procedure, each one taking different argument types.

You could similarly, using ‘interface operator’, overload the ‘+’ operator to allow

```
p3=p1+p2
```

and, if you wanted additional operators e.g. to do a two-fold rotation around the z-axis, you could have

```
p1=.z2.p1
```

where ‘.z2.’ is a newly defined operator. The procedures for overloaded operators are functions, rather than subroutines.

Pointers

One of the principles of FORTRAN is that it shouldn't be easy to fool around with memory. C programmers might appreciate being able to access and manipulate memory directly using pointers, but in principle a FORTRAN programmer can make no assumptions about how, for instance, arrays will be stored in memory.

A C pointer can be thought of as the memory address of an object; a FORTRAN pointer is best thought of as an alias, or link, to another object. A pointer can only point to an object which has a target attribute as shown below.

```

real, pointer :: p1,p2      ! will point to real target
                           ! objects

real, target  :: t1=3.4,t2=4.5 ! real variables which can
                           ! be pointed to

p1 => t1; p2 => t2          ! p1 points to t1, p2 points
                           ! to t2

print*,t1,t2,p1,p2        ! 3.4 4.5 3.4 4.5

p2 => p1                   ! p2 points to p1, and so
                           ! points to t1

t1 = 5.2
print*,t1,t2,p1,p2        ! 5.2 4.5 5.2 5.2

p2 => t2                   ! p2 points to t2
p1 = p2                   ! i.e. t1 = t2
print*,t1,t2,p1,p2        ! 4.5 4.5 4.5 4.5

```

So you can play games with pointers by pointing them at different objects. Pointers can also point to arrays, and when they are declared, only the rank of the target array needs to be given, not the sizes of any of the dimensions, as shown below.

```

real, dimension (:,:), pointer :: pa2
real, dimension (100,200,300), target :: ta3

pa2 => ta3(:, 1, 2:100:2)
! pa2(100,50) points to a subset of ta3

```

The subset of ta3 covers all of its first index, just the first element of its second index, and elements 2,4,6,...,100 of its third index. pa2 can then be treated exactly as a normal

array, except that whatever changes are made to its own elements will also be made to the corresponding elements of ta3.

Dynamic Memory Allocation

Storage can be dynamically allocated and de-allocated to pointers:

```
real, dimension (:,:), pointer :: pa2

integer :: i1,i2
...
allocate(pa2(i1,i2))      ! pa2(i1,i2) now allocated
...
deallocate(pa2)
```

And also directly to allocatable arrays:

```
real, dimension (:,:), & allocatable :: ta2
...
```

However, unlike allocatable arrays, pointers can be passed as procedure arguments:

```
p => t
call sub(p)
...
subroutine sub(x)
...
```

In the above example, x will point to t (or whatever subset of t that p is pointing to).

Whole Array Operations

It's in the area of whole array operations that FORTRAN 90 really scores. Not only can it give very concise and readable source code, it allows your compilers and vector and parallel processors to make the most of their processing potential.

Consider having to code the following example. Three, two-dimensional, arrays (of undefined but identical size) get passed to a subroutine. The first array is to be normalised to the second array, unless the element of the second array is zero, in which case it is to be set to the corresponding element of the third array.

Sound complicated?

```
subroutine norm(a1,a2,a3)
implicit none
real, dimension(:,,:), intent(inout) :: a1

real, dimension(:,,:), intent(in) :: a2,a3

where(a2 /= 0.0)    ! i.e. not equal to zero
  a1=a1/a2
elsewhere
  a1=a3
end where
return
end subroutine norm
```

The statements look as if they apply to single elements, but the where statement actually operates on every single element in the array. Single values are also automatically expanded, e.g.

```
a1=1.0
```

would set every element of a1 to 1.0.

Summary

In FORTRAN 90, FORTRAN has come a long way. It may not have answered all its critics, but for many people it will be the language of choice for numeric applications - particularly on the increasing number of parallel processors coming into being.

See also “FORTRAN 90 Explained” by Metcalf and Reid.

Tessella Support Services plc
Creating Software for Science and Engineering

Tessella's services range from feasibility studies, through system design, development, implementation and ongoing support. Our expertise includes:

Data Analysis Software
Data Capture
Simulation Software
Advanced Graphics
Systems Support
Database Applications

Other Technical Supplements available include:

- | | |
|---|--|
| <input type="checkbox"/> Archiving of Electronic Info | <input type="checkbox"/> Object Oriented Programming |
| <input type="checkbox"/> Active Server Pages | <input type="checkbox"/> Pocket PC |
| <input type="checkbox"/> Automated GUI Testing | <input type="checkbox"/> Portable GUI Development |
| <input type="checkbox"/> Bayesian Statistics | <input type="checkbox"/> Printer Technology Guide |
| <input type="checkbox"/> Beowulf Clusters | <input type="checkbox"/> Real Time Systems |
| <input type="checkbox"/> C++ | <input type="checkbox"/> Regression Testing |
| <input type="checkbox"/> Client-Server Technology | <input type="checkbox"/> Security and the Internet |
| <input type="checkbox"/> COM | <input type="checkbox"/> Simulation |
| <input type="checkbox"/> Computational Fluid Dynamics | <input type="checkbox"/> Soft Computing |
| <input type="checkbox"/> Computer Image Processing | <input type="checkbox"/> Software Design Methodologies |
| <input type="checkbox"/> Decision Support Systems | <input type="checkbox"/> Software Development Cycle |
| <input type="checkbox"/> Electronic Data Capture | <input type="checkbox"/> Software Documentation |
| <input type="checkbox"/> Electronic Lab Notebooks | <input type="checkbox"/> Software Portability |
| <input type="checkbox"/> Excel | <input type="checkbox"/> Software Re-engineering |
| <input type="checkbox"/> Extending the Life of Software | <input type="checkbox"/> Software Specification |
| <input type="checkbox"/> Federal Drug Administration | <input type="checkbox"/> SQL |
| <input type="checkbox"/> FORTRAN 90 | <input type="checkbox"/> UNIX Inter-Process Comms |
| <input type="checkbox"/> Grid Computing | <input type="checkbox"/> UNIX Systems Performance |
| <input type="checkbox"/> High Throughput Screening | <input type="checkbox"/> UNIX Workstations |
| <input type="checkbox"/> Instrumentation | <input type="checkbox"/> Visual Basic 6 |
| <input type="checkbox"/> Integrated Lab Systems | <input type="checkbox"/> WAP |
| <input type="checkbox"/> J2EE | <input type="checkbox"/> Web Services |
| <input type="checkbox"/> Java | <input type="checkbox"/> Windows 2000 Services |
| <input type="checkbox"/> Lims | <input type="checkbox"/> XML |
| <input type="checkbox"/> Linux | <input type="checkbox"/> X Windows |
| <input type="checkbox"/> Microsoft Net | |



INVESTOR IN PEOPLE

Tessella Support Services plc

3 Vineyard Chambers, Abingdon, Oxon, OX14 3PX, England

Tel: (+44) (0) 1235 555511 Fax: (+44) (0) 1235 553301

E-mail: info@tessella.com Web Address: <http://www.tessella.com>