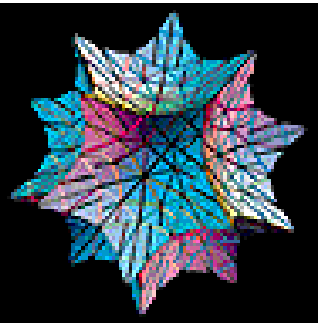


# ITC Research Computing Support Object-Oriented Programming in Fortran 95



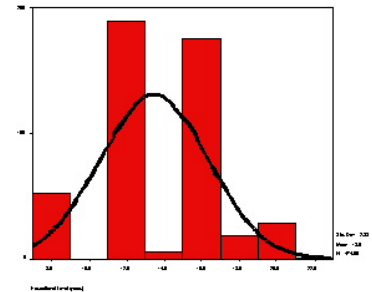
By: Katherine Holcomb

Research Computing Support Center

Phone: 243-8800 Fax: 243-8765

E-Mail: [Res-Consult@Virginia.EDU](mailto:Res-Consult@Virginia.EDU)

<http://www.itc.Virginia.edu/researchers>



# Fortran 95

"I do not know what the scientific programming language of the year 2000 will look like but it will be called Fortran"

-- Attributed to many people including  
Seymour Cray and John Backus

Fortran 95 is a minor update to Fortran 90. Fortran 90 was a major advancement of the language.

Fortran 90/95 is a fully modern programming language with a full set of constructs. No gotos required!

Fortran 95 has a particularly rich set of constructs, most of them new, for dealing with arrays. Like its predecessor Fortran 77, it has many mathematical intrinsics and complex arithmetic is automatically supported.

Free-format programming style. No more counting of columns. Indicate line continuations with ampersand (&) at the *end* of the line.

The official name of the language has been Fortran (*not* all caps) since Fortran 77.

# Some Useful New Features of F95

- Array Syntax
- Memory Management
- Interfaces
- Improved character handling
- New Intrinsic
- Kind (precision)

# Array Constructs

- Array syntax
  - dimension(10,10) :: A, B
  - A = B
  - A = sin(B)
- Inquiry
  - size(A, ndim), shape(A)
- Dynamic rearrangement of arrays
  - reshape, spread
- Operations on arrays
  - matmul(A,B)
  - dot\_product(V1,V2)
- Transformational functions
  - all, any, count, maxval, minval, product, sum

# Memory Management

- **Automatic arrays**

- pass array and its size to subroutine

```
dimension A(10,10)
```

```
n = 10
```

```
call sub(A, n)
```

```
.....
```

```
subroutine sub(A,n)
```

```
dimension A(n,n)
```

```
.....
```

- **Allocatable arrays**

```
real, dimension(:,:), allocatable :: A
```

```
allocate(A(m,n))
```

```
deallocate(A)
```

- **Pointers**

- Not true pointers in C sense, more like handles
- must have a target declared in order to be assigned (dangling not allowed)

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

```
real, dimension(10,10), target :: T
```

```
P => T
```

# Interfaces

- Function/subroutine prototypes
- If an interface is declared, the compiler will check that type, kind, and number of variables in the parameter list matches, thus eliminating a major source of bugs

interface

```
subroutine mysub(a, n, r)
  real, dimension(:, :, :), allocatable :: a
  real :: r
  integer :: n
end subroutine
```

end interface

# Character Handling

- New style of length declaration
  - `character(len=10) :: word`
- New intrinsics
  - lexical comparison
    - `Ige`, `Igt`, `Ile`, `Ilt`
  - Elementals
    - `adjustl`, `adjustr`, `index`, `len_trim`
  - Inquiry
    - `len`
  - Transformational
    - `repeat`, `trim`

# Miscellaneous New Intrinsics

- `random_number` and `random_seed`
- bit manipulation
  - `btest`, `ibset`, `iand`, `ior`, `ieor`, `ishft`, `not`, + others
- `date_and_time`
- `cpu_time`

# Kind

- Kind is a generalized precision
- User requests a kind of at least a certain number of decimal digits (for reals) and a certain exponent range (reals and integers). System matches this to its available types.

```
integer, parameter :: sp = selected_real_kind(7,20)
real (sp)           :: r, s, t
```

# Object-Oriented Programming

- What it is
  - A way to organize code
  - May enhance code reuse
- What it is not
  - A panacea for all software engineering problems
  - A religion
    - Use objects where appropriate, procedures where that works better

# The Two Key Ideas of OOP

- Organization of code into functional groups
  - Objects are *things* with well-defined behaviors
- Data hiding
  - Program units cannot access data that is internal to the object—interaction is via interfaces controlled by the programmer

# Example

- A graphics system: objects might consist of geometrical entities such as a line, an ellipse (a circle is a special case of the ellipse), a polygon, etc.
- Functions (e.g. translation, rotation, and scaling in this example) are defined that operate on each “thing.”

# Another Example

- In scientific or mathematical computing, a grid might be a candidate for an object, especially if it is adaptive (i.e. it has behaviors)
  - grid might be defined as a type
  - “behaviors” might include initialization, resizing, replication, etc.

# OOP in Fortran

- Derived Types
- Pointers
- Modules
  - Can be similar to classes
  - public/private types
- Function/Subroutine Overloading
- No inheritance (until F2003 compilers are widely available)

# Derived Types

Type person

character(len=50) :: name

real :: age

integer :: id

end type person

Like a struct in C

Functions cannot be members of a type

# Modules

- Modules are the key to OOP in Fortran
- Modules are independent code units
- Must be linked in appropriate order
- Must be compiled before any dependent units are compiled
- Access the module via the USE keyword
  - USE must be the first statement(s) after the unit declaration (program, subroutine, etc.)

```
program my_prog
  use my_mod
```

# Example Compilation

```
ifort -c my_obj.f90
```

```
ifort -c my_prog.f90
```

```
ifort -o my_prog my_prog.o my_obj.o
```

- Use make under Unix
- makemake script automatically generates dependency lists

<http://www.fortran.com/fortran/makemake.perl>

# EXAMPLE

```
module my_object
  implicit none
  integer          :: i, j, l=5
  real             :: s=25

  type obj
    integer :: m
    real, dimension(:,:), pointer :: a
  end type obj

  private:: i,j,l
  private:: s

contains
  subroutine obj_init(an_obj, n)
    integer, intent(in) :: n
    type(obj), intent(inout) :: an_obj
    allocate (an_obj%a(n,n))
    an_obj%a = s
    an_obj%m = l
  end subroutine obj_init
```

```
subroutine obj_change(an_obj, n, r)
integer, intent(in) :: n
real, intent(int) :: r
type(obj), intent(inout) :: an_obj
    an_obj%a = r
    an_obj%m = n
end subroutine obj_change
```

```
subroutine obj_del(an_obj)
type(obj), intent(inout) :: an_obj
    deallocate(an_obj%a)
end subroutine obj_del
```

```
end module my_object
```

```
program my_prog
use my_object
implicit none
```

```
integer  :: mval
real     :: rval
```

```
integer  :: ns
type(obj) :: the_obj
```

```
ns = 10
call obj_init(the_obj, ns)
```

```
print *, the_obj%m
```

```
rval = 30.
mval = 15
call obj_change(the_obj, mval, rval)
```

```
print *, the_obj%m
```

```
stop
end
```

# Function Overloading

- Functions can be overloaded to define operations on different types, including user defined types. The function can then be invoked by a single name
  - All intrinsic functions are automatically overloaded for the appropriate system-defined types, usually single precision, double precision, and complex
- Operators can be overloaded with the `OPERATOR(<operator symbol>)` keyword. The definition must use the same number of operands as the system operator
- Overloading is accomplished in a module with the `MODULE PROCEDURE` keywords

## EXAMPLE with operator overloading

```
module my_object
  implicit none
  integer          :: i, j, l=5
  real             :: s=25

  type obj
    integer :: m
    real, dimension(:,:), pointer :: a
  end type obj

  interface operator(+)
    module procedure obj_add
  end interface

  public :: obj_init, obj_change, obj_del

  private:: i,j,l
  private:: s
  private::obj_add
```

contains

```
subroutine obj_init(an_obj, n)
integer, intent(in) :: n
type(obj), intent(inout) :: an_obj
  allocate (an_obj%a(n,n))
  an_obj%a = s
  an_obj%m = l
end subroutine obj_init
```

```
subroutine obj_change(an_obj, m, r)
integer, intent(in) :: m
real, intent(in) :: r
type(obj), intent(inout) :: an_obj
  an_obj%a = r
  an_obj%m = n
end subroutine obj_change
```

```
function obj_add(obj1, obj2)
  type(obj), intent(in) :: obj1, obj2
  type(obj)             :: obj_add
  call obj_init(obj_add, size(obj1%a,1))
  obj_add%m = obj1%m + obj2%m
  obj_add%a = obj1%a + obj2%a
end function obj_add
```

```
subroutine obj_del(an_obj)
  type(obj), intent(inout) :: an_obj
  deallocate(an_obj%a)
end subroutine obj_del
```

```
end module my_object
```

```
program my_prog
use my_object
implicit none
```

```
integer          :: nval
real             :: rval
integer, parameter :: ns = 10
type(obj)       :: this_obj, that_obj, sum_obj
```

```
call obj_init(this_obj)
call obj_init(that_obj)
```

```
print *, this_obj%m
```

```
print *, that_obj%m
```

```
rval = 30.
nval = 15
```

```
call obj_change(this_obj, nval, rval)
```

```
print *, this_obj%m
```

```
sum_obj = this_obj + that_obj
```

```
print *, sum_obj%m
```

```
stop
```

```
end
```

# Advanced Use of Modules

- Can rename entities from a module (useful if two independent modules must be used but have something like a procedure with the same name)

use <module name>, <rename list>

rename list has the form

<local\_name>=><module\_name>

use stats\_lib, sprode=>prod

use math\_lib

This is not required if one (or both) of the two entities is never invoked

# Using a Subset of the Names in a Module

- USE <module name>, ONLY: [list]
  - Each name in [list] is a public entity in the module
  - Can rename as in previous example

use stat\_lib, only sprod=>prod, mult

# Other Uses for Modules

- Replace common
  - common is deprecated and awkward
- Define variables global to two or more program units
  - use sparingly, but sometimes it's necessary
  - this is also similar to common
  - kinds can be defined as global variables and used in every program unit

# Conclusions

- Organize your code around logical groupings
  - Some entities in scientific/mathematical coding are objects fairly naturally
    - Types on which operators are defined and can be overloaded are obvious candidates
  - Often related functions can be placed into a module even if they are not “objects” per se.
    - Examples might include a group of matrix manipulation routines. Once such a module is written, it can be invoked from many codes.

# Upgrading F77 to F95

- F77 is a subset of F95, so new features can be added gradually. A suggested sequence:
  1. Eliminate common, replace with modules
  2. Add array syntax, new intrinsics as appropriate
  3. Streamline memory management
  4. Analyze code, put obvious groups together into modules
  5. Reorganize, introducing objects as appropriate

# Some References

- Chivers, I. and J. Sleightholme. *Introducing Fortran 95*. Springer, 2000
- Kerrigan, J.F. *Migrating to Fortran 90*. O'Reilly & Associates 1993 (out of print)
- Metcalf, M. and J. Reid. *Fortran 90/95 Explained*. Oxford University Press 1998.