Fortran Modernization Project

Viktor K. Decyk

UCLA / Jet Propulsion Laboratory

Charles D. Norton

Jet Propulsion Laboratory

Project Goals

- **Enhance codes to benefit from modern software engineering techniques**
  - New features of Fortran 90/95 standard
  - Dynamic memory
  - Problem domain based design
  - Reorganization to promote collaborative development

Protect existing investment in software development and efficiency while benefiting from increased safety, organization, and extensibility

Technology

- **Fortran 90/95 Features Modernize Programming**

Modules
Encapsulate data and routines across program units

Interfaces
Verifies argument types in procedure calls

Array Syntax
Simplifies whole array, and array subset, operations

Use-Association
Controls access to module content

Derived Types
User-defined types supporting abstractions in programming

Pointers/Allocatable Arrays
Supports flexible/dynamic data structures

Backward compatible with Fortran 77

FOR MORE INFO...

Fortran 90 Programming. Ellis, Philips, & Lahey; Addison Wesley, 1994
http://www.cs.rpi.edu/~szymansk/oof90.html

Fortran 77 Array Types

- **Array sizes must be known at compile time**
  - Passing a previously allocated array is allowed

```
subroutine sub77(f,n_var)
  dimension f(n_var)        <--- Assumed-Size Array
  parameter(n_param=20)     <--- Parameter Known at Compile Time
  dimension g(n_param)      <--- Explicit-Shape Array
  dimension h(n_var)        <--- Dynamic Array Not Allowed
end
```

Fortran 90 Array Types

- **Dynamic arrays are permitted**
  - Fortran 77 style arrays are also permitted

```
subroutine sub90(f)
  real, dimension(:) :: f                            ← Assumed-Shaped array
  n_var = size(f)                                    ← Obtain array size
  real, dimension(size(f)) :: g                      ← Automatic array
  real, dimension(:), save, allocatable :: h         ← Allocatable array
  real, dimension(:), pointer :: p, q                ← Pointer array
  allocate(h(n_var),p(n_var))
  q => p
  …
  deallocate(h,p)
end
```

Fortran 77 Legacy Subroutine

- **Old, ugly, but fast legacy FFT**

```
subroutine fft1r(f,t,isign,mixup,sct,indx,nx,nxh)
integer isign                     sign of transform
integer indx                      size of transform
integer nx, nxh                   array dimensions
real f(nx)                        data to be transformed
complex t(nxh)                    scratch array
integer mixup(nxh)                bit reverse table
complex sct(nxh)                  sin/cos table
c rest of procedure goes here
return
```

Fortran 90 Wrapper Procedure

- **Simple FFT wrapper procedure: call fft1r(f,1)**

```
subroutine wfft1r(f,isign,indx)
real, dimension(:) :: f                    !data to be transformed
integer :: isign                           !sign of transform
integer, optional :: indx                  !size of transform
complex, dimension(size(f)/2) :: t         !scratch array
integer, dimension(:), allocatable, save :: mixup   !bit rev table
complex, dimension(:), allocatable, save :: sct     !sin/cos table
integer, save :: nx, nxh, indx_saved = 0
    nx = size(f); nxh = size(f)/2
    if (present(indx)) then                !initialize first time
        allocate(mixup(nxh), sct(nxh)) ; indx_saved = indx
        call fft1r(f,t,0,mixup,sct,indx,nx,nxh) !create tables
    else if (indx_saved > 0) then          !perform FFT
        call fft1r(f,t,isign,mixup,sct,indx_saved,nx,nxh)
    end if
end subroutine wfft1r
```

Fortran 90 Interface Block

- **Interface statements verify argument types in procedure calls**

```
interface
    subroutine fft1r(f,t,isign,mixup,sct,indx,nx,nxh)
    integer :: isign, indx, nx, nxh
    real, dimension(nx) :: f
    complex, dimension(nxh) :: t, sct
    integer, dimension(nxh) :: mixup
    end subroutine fft1r
end interface
```

Fortran 90 Modules

- **Modules are containers for grouping type definitions, interfaces, data, and procedures**

```
module fft1r_mod
  interface                                                    <-- interfaces
     subroutine fft1r(f,t,isign,mixup,sct,indx,nx,nxh)
     ...
     end subroutine fft1r
  end interface                                     data
  integer, save :: num_errors = 0    ! keep track of errors
contains
  subroutine wfft1r(f,isign,indx)                              <-- procedures
     ...
  end subroutine wfft1r
end module fft1r_mod
```

Fortran 90 Main Program

- **Using a module makes its content available to the program unit**

```
program main
use fft1r_mod
implicit none
integer :: indx = 7
real, dimension(:), allocatable :: data
  allocate(data(2**indx))
  call wfft1r(f,0,indx)           ! initialize FFT
  call wfft1r(f,1)                ! perform FFT
end program main
```

Fortran 90 Wrapper Family

- **FFT "component": call fft1_init(indx)**

```
module fft1r_mod
integer, private, save :: indx_saved = 0
integer, dimension(:), allocatable, private, save :: mixup
complex, dimension(:), allocatable, private, save :: sct
contains
subroutine fft1_init(indx)
integer :: indx, nx, nxh
real, dimension(2**indx) :: f ; complex, dimension(2**indx/2) :: t
nx = 2**indx, nxh=nx/2
   allocate(mixup(nx),sct(nx)) ; indx_saved = indx
   call fft1r(f,t,0,mixup,sct,indx,nx,nxh) !create tables
end subroutine fft1_init
subroutine do_fft1r(f,isign)
...
```

Fortran 90 Wrapper Family

- **FFT "component": call do_fft1r(f,1)**

```
...
subroutine do_fft1r(f,isign)
integer :: isign                              ! sign of transform
real, dimension(:) :: f                       ! data to be transformed
complex, dimension(size(f)/2) :: t            ! scratch array
integer, save :: nx=size(f), nxh=size(f)/2
    if (indx_saved > 0) then                  ! perform FFT
        call fft1r(f,t,isign,mixup,sct,indx_saved,nx,nxh)
    end if
end subroutine do_fft1r
subroutine fft1r_end                          ! free dynamic storage
    deallocate(mixup,sct); indx_saved = 0
end subroutine fft1r_end
end module fft1r_mod
```

Fortran 90 Main Program

- **Using a module makes its public content available to the program unit**

```
program main
use fft1r_mod
implicit none
integer :: indx = 7
real, dimension(:), allocatable :: data
  allocate(data(2**indx))
  call fft1r_init(indx)          ! initialize FFT
  call do_fft1r(f,1)             ! perform FFT
  call fft1r_end()               ! deallocate private tables
end program main
```

Fortran 90 Derived Type

- **Derived types allow related variables to be grouped together**

```
module graphic_mod
type graphic                          ! Variables describe plot properties
    real :: xmin, xmax
    integer :: nx, isc, ist, mks
end type
contains
    subroutine DISPR(f,label,gc,error)  ! Plot array f
    implicit none
    real, dimension(:) :: f
    character(len=*) :: label
    type (graphic), intent(in) :: gc
    integer :: error
    call DISPR(f,label,gc%xmin,gc%xmax,gc%isc,gc%ist,gc%mks,&
                gc%nx,size(f),error)
    end subroutine
end module graphic_mod
```

Converting to Dynamic Memory

- **Static Memory**

```
C Original include file of common data
  PARAMETER (mdttl = 128)
  INTEGER nElt,RayID(mdttl,mdttl), …
  COMMON /EltInt/ nElt, RayID, …
```

- **Dynamic Memory**

```
! Module for common data
  module elt_common
    implicit none ; save
    INTEGER :: nElt, mdttl = 128
    INTEGER, allocatable,
                dimension(:,:) :: RayID
  contains
     subroutine new_elt_commmon()
     allocate(RayID(mdttl,mdttl),...)
     end subroutine new_elt_common
  end module elt_common
```

```
! Dynamic allocation
  program macos
    use elt_common
    call new_elt_common()
    …
  end program macos
```

Fortran 90 Wrapper (Ver. 1)

- **A very simple wrapper for a particle push**

```
subroutine wpush1(part,force,qbm,wke,dt)
real, dimension(:,:) :: part
real, dimension(:) :: force
integer :: ndim, nparticle, nx
ndim = size(part,1); nparticle = size(part,2)
nx = size(force)
  call push1(part,force,qbm,wke,ndim,nparticle,nx,dt)
end subroutine wpush1
```

Fortran 90 Wrapper (Ver. 2)

- **Encapsulate particle arguments within a derived type**

```
type species
  real, dimension(:,:), pointer :: coords
  real :: charge_to_mass, kinetic_energy
end type species
subroutine wpush1(particle,force,dt)
type (species) :: particle
real, dimension(:) :: force
real :: dt, qbm, wke
integer :: ndim, nparticle, nx
   ndim = size(particle%coords,1)
   nparticle = size(particle%coords,2) ; nx = size(force)
   qbm = particle%charge_to_mass ; wke = particle%kinetic_energy
   call push1(particle%coords,force,qbm,wke,ndim,nparticle,nx,dt)
end subroutine wpush1
```

Fortran 90 Class Structure

- **Classes group data, types, and procedures**

```
module plasma_class
  type species
     real, dimension(:,:), pointer :: coords
     real :: charge_to_mass, kinetic_energy
  end type species
contains
  subroutine new_species(this,ndim,nparticle,qbm)  ! Constructor
  type (species) :: this
  integer :: ndim, nparticle ; real :: qbm
     allocate(this%coords(ndim,nparticle))
     this%charge_to_mass = qm ; this%kinetic_energy = 0.
! call some procedure to assign initial coordinates here ...
  end subroutine new_species
  subroutine wpush1(particle,force,dt) ...
end module plasma_class
```

Fortran 90 Plasma Program

- **Program now uses abstractions from the problem domain**

```
program main
use plasma_class
implicit none
integer :: nsize = 128
type (species) :: electrons
real, dimension(:), allocatable :: force
  allocate(force(nsize))
  call new_species(electrons,6,60000,qbm=-1.)
  ...
  call wpush1(electrons,force,dt=.2)
  ...
end program main
```

Self Describing Objects

- **Encapsulating properties in sophisticated types**

```
module fields
use fft_module
use display_module
type field
  private
  real, dimension(:), pointer :: data
  integer :: property
  type (fftparams), pointer :: fparms
  type (displayparams), pointer :: dparms
end type field
type fftparams
  private
  integer :: indx
  integer, dimension(:), pointer :: mixup
  complex, dimension(:), pointer :: sct
end type fftparams
integer, parameter :: FFTABLE = 1, DISPLAYABLE = 2
...
```

Creating Self Describing Objects

- **A field constructor**

```
...
contains
  subroutine new_field(this,dsize,indx,dtype)
  type (field) :: this
  integer, intent(in) :: dsize
  integer, optional :: indx, dtype
  allocate(this%data(dsize))
  this%property = 0
  nullify(this%fparms); nullify(this%dparms)
  if (present(indx)) then
     this%property = this%property + FFTABLE
     call new_fftparams(this%fparms,indx)
  endif
  if (present(dtype)) then
     this%property = this%property + DISPLAYABLE
     call new_displayparams(this%dparms,dtype)
  endif
  end subroutine new_field
end module fields
```

Using Self Describing Objects

- **A main program**

```
program main
use fields
type (field) :: f
  call new_field(f,dsize=130,indx=7,dtype=ONE-D)
  ...
  call fft(f,isign=1)
  ...
  call display(f,dstyle=LINE)
  ...
end program main
```

OO Concepts in a Nutshell

- **Encapsulation With User-Defined Types**
  – Allows one to combine into a single structure related data which can be passed together to procedures.
  – Internal details of the structure can be changed without impacting the clients (users).
- **Classes**
  – Contain user-defined types and procedures that work on them.
- **Inheritance**
  – Allows a family of similar types to share common code. This family must have a special data relationship where the parent “fits” inside the child.
- **Run-Time Polymorphism**
  – Allows one to write code for a family of types, where the actual type will be determined at run-time.

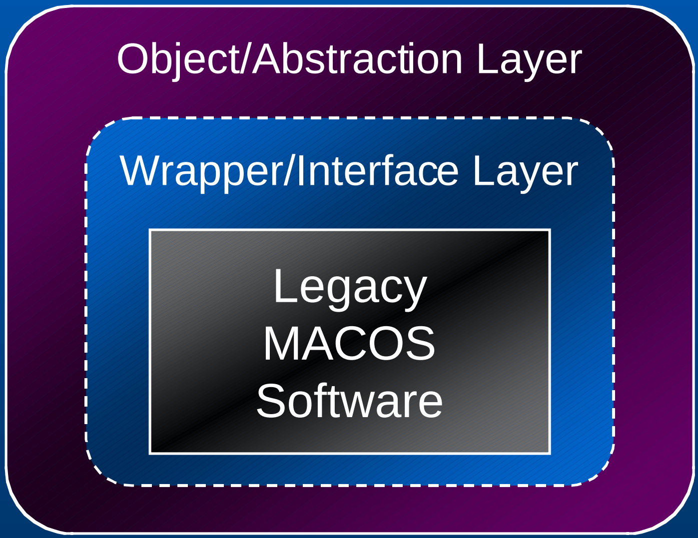
Modernizing MACOS

- **Modeling and Analysis for Controlled Optical Systems (MACOS), an important NASA code**
- **Preserve existing code, yet transform for new development**
  - Bring MACOS up to the Fortran 90/95 standard
  - Create interface layer to original code
  - Support abstraction-based programming via interfaces
  - Gradual and selective replacement of data structures
- **Benefits**
  - Code remains in use during modification

Important as more ambitious codes are developed and maintained

Development

- **Efficient interaction among MACOS, interfaces, abstraction layer, and user I/O**



Allows safe interaction with a legacy code

Process Summary

- **Legacy codes still have value, but extending that functionality has become more important**
- **Modern codes require...**
  - Greater complexity and multiple authors
  - Dynamic features and flexible design
- **Build modern superstructure while code remains in use**
  - Data abstraction and information hiding are key to limiting exposure of unnecessary details
  - Modern language features reduce inadvertent errors
- **Wrappers can extend functionality**
  - Verify preconditions, measure performance, etc...

Quotes from Stroustrup (C++ Designer)

- "More good code has been written in languages denounced as "bad” than in languages proclaimed "wonderful" - much more."
- "It would be nice if every kind of numeric software could be written in C++ without loss of efficiency, but unless something can be found that achieves this without compromising the C++ type system it may be preferable to rely on Fortran ..."
- "Fortran is harder to compete with. It has a dedicated following who... care little for programming languages or the finer points of computer science. They simply want to get their work done."
- "I see C++ as a language for scientific computation and would like to support such work better than what is currently provided. The real question is not "if?" but "how?"
- "C++ was designed to be a systems programming language and a language for applications that had a large systems-like component."
- "I am not among those who think that a single language should be all things to all people."