User defined data types (derived data types)

Fortran 90 allows object oriented programming which was not included in the earlier revisions of Fortran. Familiar to C and C++ programmers structures and classes. A user may defined a new data type which is derived from the build-in data types. A general construct is

```
type new_type_name

    type :: var1
    type :: var2
    ...

end type new_type_name
```

where `type ... end type` defines the block which derives a new data type. `new_type_name` is an arbitrary and unique name for this type. In the body `type` means one of the previously defined types (either build-in or derived). New variable of this type can be now created

```
type(new_type_name) :: variable
```

and its members can be accessed using `%` operator. For example

```
type matrix_element

    integer :: row_idx
    integer :: col_idx

    real :: data_value

end type matrix_element
```

defines new derived type called `matrix_element` and is a coordinate representation of a single matrix element with `row_idx` and `col_idx` defining its positions and `data_value` holding the real value. This object can be created as

```
type(matrix_element) :: single_element
```

or

```
type(matrix_element), dimension(:), allocatable :: matrix
```

All data creation parameters (for definition of arrays, memory allocations) also apply to derived types. Members of the created object may be accessed as

```
single_element%row_idx = 1
single_element%col_idx = 1
single_element%data_value = 3.14
```

Values can be also assigned as

```
single_element = matrix_element(1, 2, 3.14)
```

Restriction of types was that they could not include allocatable arrays in Fortran 90/05. This has been however removed in Fortran 2003. Fortran 2003 also allows more flexible assignment of data elements

```
single_element = matrix_element(row_idx = 1, col_idx = 2, data_element = 3.14)
```

or only on selection of the members

```
single_element = matrix_element(row_idx = 1, col_idx = 2)
```

Default values of the members may be also defined, in that case we could have

```
type matrix_element

    integer :: row_idx = 0
    integer :: col_idx = 0

    real :: data_value = 0.0

end type matrix_element
```

Example program

```
program derived_type

  implicit none

  type Matrix

    integer :: num_cols = 0
    integer :: num_rows = 0

    real, dimension(:,:), allocatable :: elements

  end type Matrix

  integer :: n
  integer :: i, j
  type(Matrix) :: A, B

  n = 4

  A%num_cols = n
  A%num_rows = n
  allocate( A%elements(n,n) )

  A%elements = 1.0

  do i=1, n
   print *,( A%elements(i,j), j=1, n)
  end do

  deallocate( A%elements )

end program derived_type
```

which returns

```
   1.00000000       1.00000000       1.00000000       1.00000000
   1.00000000       1.00000000       1.00000000       1.00000000
   1.00000000       1.00000000       1.00000000       1.00000000
   1.00000000       1.00000000       1.00000000       1.00000000
```

Example of allocatable array of derived type

```
program derived_type
```

```fortran
  implicit none

  type Matrix

    integer :: num_cols = 0
    integer :: num_rows = 0

    real, dimension(:,:), allocatable :: elements

  end type Matrix

  integer :: n, matrix_size
  integer :: i, j, k
  type(Matrix), dimension(:), allocatable :: A

  n = 4

  allocate( A(n) )

  do i=1, n

    allocate( A(i)%elements(n,n) )

    A(i)%num_cols = i
    A(i)%num_rows = i

    A(i)%elements = real(i)

  end do

  do i=1, n
    print *,'Element ', i

    do j=1, A(i)%num_rows
      print *,( A(i)%elements(j,k), k=1, A(i)%num_cols)
    end do

  end do

  do i=1, n
    deallocate( A(i)%elements )
  end do

  deallocate( A )

end program derived_type
```

Modules

Fortran 90 introduced construct that allows logical organization of the program, definition of semi-global variables, functions and subroutines. The construct is called `module` and has a general structure

```
module module_name

    [ body 1 ]

    contains

    [ body 2 ]

end module module_name
```

module can be used in the body of the program by giving

```
use module_name
```

command at the top of the program, function or subroutine. The variables and procedures placed in the module remain not-available until the module is used with the `use` command.

In the definition above `[body 1]` is a place for variable definition which will be semi-global, i.e. available to all functions and procedures defined in the module as global variables and to all parts of the main program that `use` the module.

`[body 2]` is place for definition of functions and subroutines which remain local to that module and all parts of program that `use` it.

Example usage

```
module square_matrix_operations

  implicit none

  contains

    subroutine set_to(A,n, val)
      implicit none
      integer, intent(in) :: n
      real, dimension(n,n), intent(inout) :: A
      real, intent(in) :: val
      A = val
    end subroutine set_to

    subroutine print_matrix(A,n)
      implicit none
      integer, intent(in) :: n
      real, dimension(n,n), intent(in) :: A
      integer :: i, j

      do i=1, n
        print *,( A(i,j), j=1,n)
      end do
    end subroutine print_matrix

end module square_matrix_operations

program module_example

  use square_matrix_operations

  implicit none

  integer :: n
  real, dimension(:,:), allocatable :: A

  n = 5
  allocate( A(n,n) )

  call set_to(A,n,1.0)
  call print_matrix(A,n)

  deallocate( A )

end program module_example
```

Procedures on derived types

Derived type may have associated functions or subroutines which always refer to the derived type variable. Functions and subroutines may be assigned as `procedures` and the call or function invocation may be renamed and always used in connection to derived variable. General definition is

```
type new_type_name

    type :: var1
    type :: var2
    ...

    contains

        procedure :: short_name => long_name
        ...

end type new_type_name
```

in the example above we have defined function/subroutine `short_name` which is defined in the body of the program as `long_name` and connected it with derived type `new_type_name`. In the body of `long_name` we need to refer to type `new_type_name` not as `type` but as `class`.

Comple example may look like this

```
module algebra

  implicit none

  type Matrix

    integer :: num_cols = 0
    integer :: num_rows = 0

    real, dimension(:,:), allocatable :: elements

    contains

      procedure :: create => create_matrix
      procedure :: delete => delete_matrix
      procedure :: print => print_matrix

  end type Matrix
```

```
  contains
   subroutine create_matrix(A,nrow,ncol)

      implicit none
      class(Matrix), intent(inout) :: A
      integer, intent(in) :: nrow, ncol

      A%num_cols = ncol
      A%num_rows = nrow

      allocate( A%elements(nrow,ncol) )
      A%elements = 0.0

    end subroutine create_matrix

    subroutine delete_matrix(A)

      implicit none
      class(Matrix), intent(inout) :: A

      A%num_cols = 0
      A%num_rows = 0
      deallocate( A%elements )

    end subroutine delete_matrix

    subroutine print_matrix(A)

      class(Matrix), intent(in) :: A
      integer :: i, j

      do i=1, A%num_rows
        print *,( A%elements(i,j), j=1, A%num_cols )
      end do

    end subroutine print_matrix

end module algebra

program derived_type

  use algebra

  integer :: n
  integer :: i, j
```

```
  type(Matrix) :: A, B

  n = 4

  call A%create(n,n)
  call A%print
  call A%delete

end program derived_type
```

Overloading operators

To complete the definition of derived data types and provide more object-oriented programming, Fortran allows for overloading of operators and new definitions. In the example below, operator `*` is overloaded and suppose to act on `integer - real` pair

```
interface operator (*)

    real function F1(x,y) result(val)
        integer, intent(in) :: x
        real, intent(in) :: y
    end function F1

    real function F2(y,x) result(val)
        integer, intent(in) :: x
        real, intent(in) :: y
    end function F2

end interface

integer :: i
real :: x, r

x = i * r  ! Fortran uses F1
x = r * i  ! Fortran uses F2
```

Both functions `F1` and `F2` need to be defined in the body of the program. The same may be done on derived types, for example