Technical University of Denmark

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

High Performance Computing

FORTRAN, OpenMP and MPI

41391

Content

Fortran 2003/2008 extensions:

- Floating point exception handling.
- Type parameters and procedure pointers:
    - Parameterized derived types.
    - Abstract interfaces.
- Object-oriented programming:
    - Type extension.
    - Polymorphic entities.
    - Type bound procedures.
- Submodules.

Floating point exception

Floating point exception

- Most computer hardware is based on the IEEE (and ISO) standard for binary floating-point arithmetic.
- The standard defines:
  - **arithmetic formats**: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
  - **interchange formats**: encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form.
  - **rounding rules**: properties to be satisfied when rounding numbers during arithmetic and conversions.

https://en.wikipedia.org/wiki/IEEE_754

Floating point exception

- The standard defines (cont.):
  - **operations**: arithmetic and other operations on arithmetic formats.
  - **exception handling**: indications of exceptional conditions (such as division by zero, overflow, etc.)
- The standard also includes extensive recommendations for advanced exception handling, additional operations (such as trigonometric functions), expression evaluation, and for achieving reproducible results.

Floating point exception

- FORTRAN IEEE floating point *exception handling* since FORTRAN 2003:
  - The standard specifies the possibility of exceptions being trapped by user-written handlers, but this **inhibits optimization** and is not supported by the FORTRAN features.
  - FORTRAN 2003 support the possibility of halting the program **AFTER** the exception has signaled.

Floating point exception

- **Overflow**: occurs if the exact result of an operation with two normal values is too large for the data format. The stored result is: -∞, HUGE(x), -HUGE(x), or ∞ according to the rounding mode.
- **divide_by_zero:** occurs if a finite nonzero value is divided by zero. The stored result is -∞ or ∞ with the correct sign.
- **Invalid:** occurs if the operation is invalid, fx. $-\infty \ast 0$, $0/0$, or when an operand is a signaling **NaN** (Not a Number).

Floating point exception

- **Underflow:** occurs if the result of an operation with two finite nonzero values cannot be represented exactly, and it too small to represent with full precision.
- **Inexact:** occurs if the exact result of an operation cannot be represented in the data format without rounding.
- The IEEE features are accessed through the *modules*:
  - ieee_exceptions.
  - ieee_arithmetic.
  - ieee_features.
- The use of the ieee_features MODULE affects the way the code is compiled in the scoping unit (equivalent to and replacing a compiler directive).

Floating point exception

- If a scoping unit does not access ieee_exception or ieee_arithmetic, then the level of IEEE support is processor dependent, and need not include support for any exceptions.
- The module ieee_features contains the derived type: ieee_feature_type, that identifies a particular feature.

  Example:

  ```
  USE, INTRINSIC :: ieee_features, ONLY : ieee_datatype
  ```

  will ensure that the scoping unit will be compiled supporting (slower) code for IEEE arithmetic for at least one kind of real.

Floating point exception

- The complete list of named constants are:
    - ieee_datatype: control of IEEE arithmetic.
    - ieee_denormal: control of denormalized numbers.
    - ieee_divide: IEEE divide.
    - ieee_halting: control of halting for each flag.
    - ieee_inexact_flag: inexact exception.
    - ieee_inf: support $-\infty$ and ∞
    - ieee_invalid_flag: invalid flag.
    - ieee_nan: NaN.
    - ieee_rounding: rounding mode all 4 modes.
    - ieee_sqrt: IEEE square root.
    - ieee_underflow_flag: underflow exceptions.

Floating point exception

- There are five FORTRAN exception flags which are either *signaling* or *quiet*:
  - ieee_invalid.
  - ieee_overflow.
  - ieee_divide_by_zero.
  - ieee_underflow.
  - ieee_inexact.
- The value may be inquired by the function:

```
CALL ieee_get_flag(flag,flag_value)
```

flag (IN) is of type: type(ieee_flag_type).
flag_value (OUT) is either true (signaling) or false (quiet).

Floating point exception

- Some processors allow control during program execution of whether to abort or continue execution after an exception has occurred.

- Halting is controlled by the elemental subroutines:

```
    CALL ieee_get_halting_mode(flag,halting)
    CALL ieee_set_halting_mode(flag,halting)
```

  where halting is either true or false.

Floating point exception

If a flag is signaling on entry to a procedure, the processor will set it to quiet on entry and restore it to signaling on return. This allows exception handling within the procedure to be independent of the state of the flag on entry, while retaining their “sticky” properties.

- A sticky flag remain signaling until explicitly set quiet:

```
CALL ieee_set_flag(flag,flag_value)
```

flag (IN) is of type: type(ieee_flag_type).
flag_value (IN) is either true (signaling) or false (quiet).

Floating point exception

- The floating point status is stored by

  CALL ieee_get_status(status)

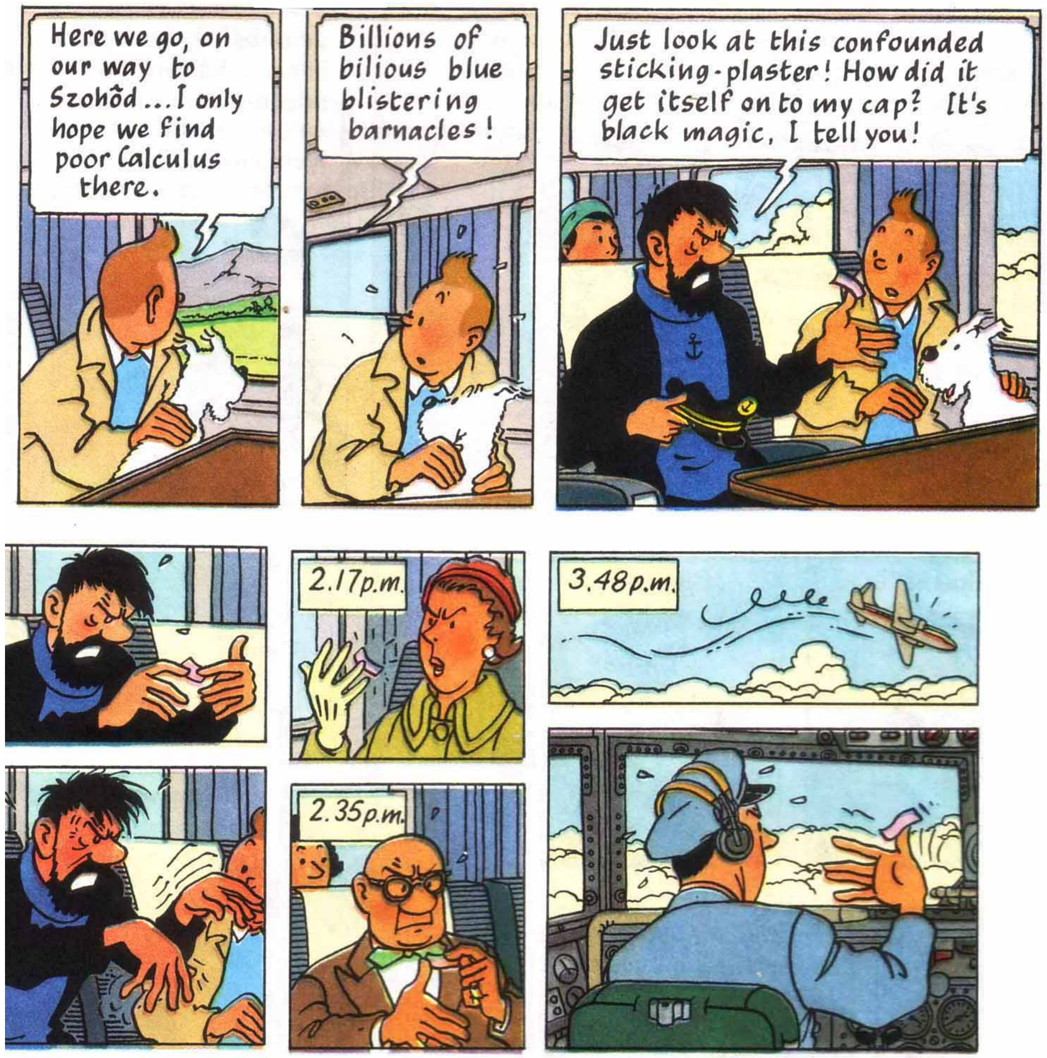
  status intent is OUT – it returns the floating-point status, including all the exception flags, the rounding mode, and the halting mode.

- The floating point status is restored

  CALL ieee_set_status(status)

  status intent is IN and set previously by ieee_get_status()

Floating point exception


Here we go, on our way to Szohôd ... I only hope we find poor Calculus there.
Billions of bilious blue blistering barnacles!
Just look at this confounded sticking-plaster! How did it get itself on to my cap? It's black magic, I tell you!
2.17p.m.
3.48p.m.
2.35p.m.

Floating point exception

```fortran
! from: http://www.nag.co.uk/nagware/np/doc/faq.asp
Program turn_underflow_flag_off
  Use Ieee_Exceptions
  type(IEEE_STATUS_TYPE) status
  logical :: fail

  Implicit None

  if (.not.Ieee_support_halting(Ieee_underflow)) then
     print*,'compiler does not support halting; exiting!'
     stop
  else
     print*,'compiler supports halting!'
  endif

  Call Ieee_Get_Status(status)
  Call Ieee_Set_Halting_mode(Ieee_underflow, .false.)

  Call make_underflow(0.5)

  Call Ieee_Get_Flag(Ieee_Underflow, fail)
  print*,'fail ? ',fail
  Call Ieee_Set_Status(status)
Contains
```

```fortran
  Subroutine make_underflow(x)
    Implicit none
    Real, Intent(In) :: x
    Real :: y
    Integer :: n
    y = x
    n = 1
    Do
      y = y**2
!     If (y< 1e-4) Exit
      If (y==0) Exit
      n = n + 1
      Print *, 'Step', n, 'value', y
    End Do
    Print *, 'Zero reached at step', n
  End Subroutine
End Program
```

Type parameters and procedure pointers

Deferred type parameters

- For characters, the length can be *deferred* until allocation of pointer association:

  ```
  character(len=:), pointer :: varchar
  character(len=100), target :: name
  character(len=200), targer :: address
  …
  varchar => name
  …
  varchar => address
  ```

- The length of varchar after each pointer assignment is the same as that of the target.
- For intrinsic types ONLY characters can be deferred.

Type parameters inquiry

- The (current) value of a type parameter of a variable can be discovered by a *type parameter inquiry*

  Example:

  ```
  real(selected_real_kind(10,20)) :: z(100)
  character(len=100) :: ch
  …
  print*,z%kind
  print*,ch%len
  ```

- **NOT SUPPORTED BY STUDIO 12.5, GFORTRAN 5.4**
- **SUPPORTED BY INTEL 15 (and higher)**

Parameterized derived type

- *Type parameters* have been introduced for derived types, similar to type parameters of intrinsic types.
- Two types of type parameters:
  - *kind* parameters (must be known at compile time).
  - *len* parameters (defined at run time).

Example:

```
type matrix(real_kind, n,m)
      integer, kind    :: real_kind
      integer, len     :: n,m = 4 ! m = 4 is default
      real(real_kind) :: value(n,m)
end type matrix
integer, parameter :: mk = kind(1.0d0)
type (matrix(mk,3,4)) :: p
type (matrix(mk,m=3)) :: r
type (matrix(m=4,n=6,real_kind=mk) :: q
```

Abstract interfaces

- An abstract interface gives a name to a set of characteristics and argument keyword names that would constitute an explicit interface to a procedure – without declaring any actual procedure to have those characteristics.
- Example:

```
abstract interface
    subroutine boring_sub_without_args
    real function r2_to_r(a,b)
        real, intent(in) :: a,b
     end function r2_to_r
end interface

procedure(boring_sub_without_args) :: sub1,sub2
procedure(r2_to_r) :: modulus, xyz
```

Abstract interfaces

- The abstract interface name may be used in the *procedure* statement to declare procedures which might be procedures, dummy procedures, procedure pointers, or deferred type-bound procedures.
- The *procedure* statement can be used with any procedure that has an explicit interface.
- Example: If fun has an explicit interface, then

  ```
  procedure (fun) :: fun2
  ```

  declared fun2 to be a procedure with an identical interface to that of fun.
- The procedure statement is not available for generic procedures (fx. cos(x)) but for specific procedures that is a member of a generic set (fx. dcos(x)).

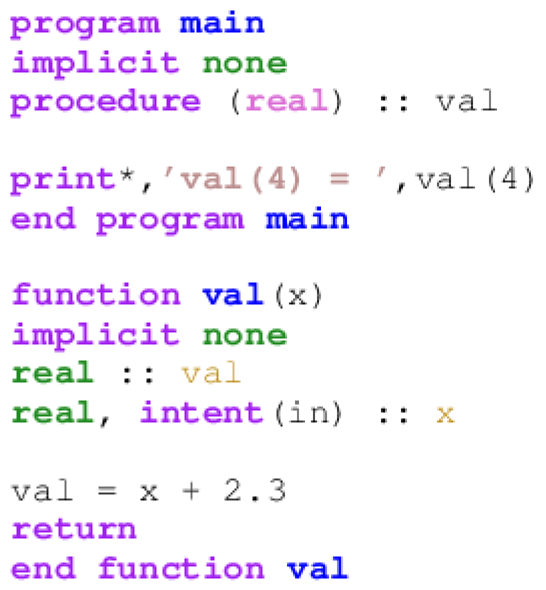
Abstract interfaces

- The procedure statement can be used to declare procedures with implicit interfaces.
- Example:

  procedure () :: x
  procedure (real) :: y

  equivalent to:

  external :: x
  real, external :: y



```
program main
implicit none
procedure (real) :: val

print*,'val(4) = ',val(4)
end program main

function val(x)
implicit none
real :: val
real, intent(in) :: x

val = x + 2.3
return
end function val
```

Procedure pointers

- Procedure pointers are pointers to procedures and may have implicit or explicit interfaces.
- Example:

```
pointer :: sp
interface ! explicit interface to subroutine
   subroutine sp(a,b)
      real, intent(inout)      :: a
      real, intent(in)     :: b
   end subroutine sp
end interface
real, external, pointer :: fp ! implicit interface to function
fp => fun
sp => sub
print*,fp(x) ! prints fun(x)
call sp(a,b) ! calls sub
```

Object-oriented programming

Type extension

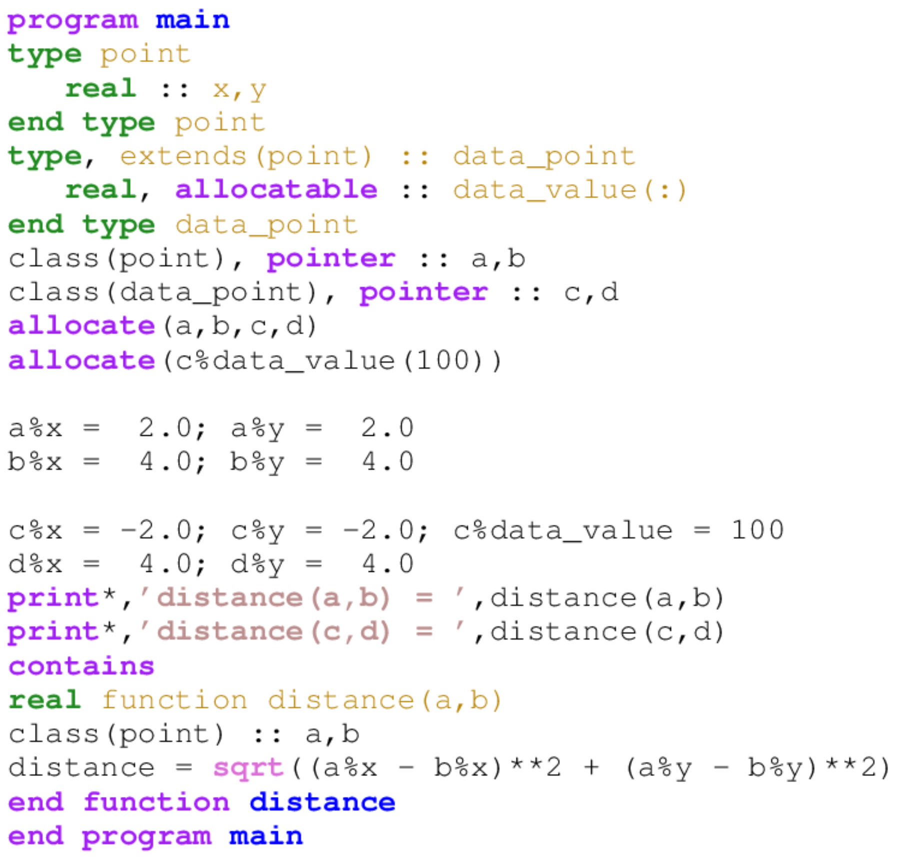
- Type extension creates new derived types by extending pre-existing derived types.
  Example:

```
TYPE person
   CHARACTER(LEN=256) :: name
   REAL :: age
   INTEGER :: id
END TYPE person
TYPE, EXTENDS(person) :: employee
    INTEGER :: CPR
    REAL :: salary
END TYPE employee
TYPE (employee) :: director
PRINT*,director%name,director%person%name
```

  returns the same.

Polymorphic entities

- FORTRAN 2003 supports polymorphic variables using the class keyword (data type may vary at run time).
  Example:



```fortran
program main
type point
   real :: x,y
end type point
type, extends(point) :: data_point
   real, allocatable :: data_value(:)
end type data_point
class(point), pointer :: a,b
class(data_point), pointer :: c,d
allocate(a,b,c,d)
allocate(c%data_value(100))

a%x =  2.0; a%y =  2.0
b%x =  4.0; b%y =  4.0

c%x = -2.0; c%y = -2.0; c%data_value = 100
d%x =  4.0; d%y =  4.0
print*,'distance(a,b) = ',distance(a,b)
print*,'distance(c,d) = ',distance(c,d)
contains
real function distance(a,b)
class(point) :: a,b
distance = sqrt((a%x - b%x)**2 + (a%y - b%y)**2)
end function distance
end program main
```

Type-bound procedures

- Invoke procedure to perform a task whose nature varies according to the dynamic type of a polymorphic object.
- Type-bound procedures are invoked through an object, and the actual procedure executed depends on the dynamic type of the object.
- In other languages type-bound procedures are know as *methods*.
- Type-bound procedures are defined by introducing a 'contains' in the type definition declaration.

Type-bound procedures

```
module mytype_module
   type mytype
      private
      real :: myvalue(4) = 0.0
      contains
      procedure :: write => write_mytype
      procedure :: reset
   end type mytype
   private :: write_mytype, reset
   contains
   subroutine write_mytype(this,unit)
   class(mytype)      :: this
   integer, optional :: unit
   if (present(unit)) then
      write(unit,*) this%myvalue
   else
      print*,this%myvalue
   endif
   end subroutine write_mytype
   subroutine reset(variable)
      class(mytype) :: variable
      variable%myvalue = 0.0
   end subroutine reset
end module mytype_module
```

```
     program main
     use mytype_module
     type(mytype) :: x
! works as long as we use it on mytype
     call x%write(6)
     call x%write
     call x%reset

! below does not work because the subroutines are private
!    call write_mytype(x,6)
!    call reset(x)
     end program main
```

Generic type-bound procedures

```
module container_module
   private
   type, public :: container
      integer, private :: i = 0
      complex, private :: c = (0.,0.)
   contains
      private
      procedure :: xi => extract_integer_from_container
      procedure :: xc => extract_complex_from_container
      generic, public :: extract => xi, xc
   end type
   contains
      subroutine extract_integer_from_container(this,val)
         class(container) , intent(in) :: this
         integer, intent(out)          :: val
         print*,'calling extract with an integer argument'
         val = this%i
      end subroutine extract_integer_from_container

      subroutine extract_complex_from_container(this,val)
         class(container) , intent(in) :: this
         complex, intent(out)          :: val
         print*,'calling extract with a complex argument'
         val = this%c
      end subroutine extract_complex_from_container

end module container_module
```

```
program main
   use container_module
   type(container)  :: v
   integer :: ix
   complex :: cx

   call v%extract(ix)
   call v%extract(cx)
end program main
```

Submodules

- Modules in Fortran 95 may create problems for very large programs. The module encapsulation may result in very large source code and repeated recompilation of program lines that effectively did not change.
- To circumvent this Fortran 2008 introduced *submodules.*
- Example:

```
program main
use points
type(point) :: p1,p2

p1%x = 1.0
p1%y = 1.0
p2%x = 0.0
p2%y = 0.0

print*,'point_dist = ',point_dist(p1,p2)

end program main
```

```
module points
   type :: point
      real :: x,y
   end type
   interface
     real module function point_dist(a,b)
     type (point), intent(in) :: a,b
     end function point_dist
   end interface
end module points

submodule (points) points_a
contains
  real module function point_dist(a,b)
  type(point), intent(in) :: a,b
  point_dist = sqrt((a%x-b%x)**2 + (a%y-b%y)**2)
  end function point_dist
end submodule points_a
```