Karlsruher Institut für Technologie

Fortran 95/2003 Course

Fortran 2003 by Hartmut Häfner
March 26, 2015

STEINBUCH CENTRE FOR COMPUTING - SCC

Literature

- Official home of Fortran Standards
  http://www.nag.co.uk/sc22wg5/

- The New Features of Fortran 2003, John Reid,
  WG5 Convener, JKR Associates, 24 Oxford Road Benson,
  Oxon OX10 6LX, UK, jkr@rl.ac.uk
  ftp://ftp.nag.co.uk/sc22wg5/n1551-n1600/n1579.pdf
  (gives an overview over the new features, many examples are taken from this article)

- The new features of Fortran 2008, John Reid, June 13 2008
  ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1735.pdf

- Fortran 95/2003 Explained, M. Metcalf, J. Reid and M. Cohen
  2004, Oxford University Press

Fortran 2003: Important new Features

- **A large number of miscellaneous additions**
- **Access to run time environment**
- **Enhancements for modules**
- **Interoperability with the C programming language**
- **Support for IEEE floating-point arithmetic and exceptions**
- Functionality enhancements for allocatable arrays
- Pointer enhancements
- Derived type enhancements
- Support for Object-oriented programming
- IO enhancements

Miscellaneous Enhancements

- Longer names and statements
- Additional characters (~, \, [,], {, }, |, #, @)
  ((/a,b,c/) → [a,b,c])
- Support for the 4 byte universal character set (UCS-4); the standard encoding of UCS-4 is supported (UTF-8)

```
open(unit,"file.txt",encoding=„utf-8")
```

- Initialization expressions are generalized
- New attributes: volatile, asynchronous
  - volatile: using a volatile variable means reloading it from memory, can be used in order to prevent compiler optimization
- Generalized array constructor - strings of different length are allowed in an array constructor

Access to Run Time Environment

- New intrinsic module: `ISO_FORTRAN_ENV`
  - `INPUT_UNIT` for the asterisk in read(*,fmt=....)
  - `OUTPUT_UNIT` for the asterisk in write(*,fmt=....)
  - `ERROR_UNIT` error unit
  - `IOSTAT_END` holds end-of-file result of IOSTAT
  - `IOSTAT_EOR` holds end-of-record result of IOSTAT
  - `FILE_STORAGE_UNIT` unit of measurement of recl= in bits
  - `CHARACTER_STORAGE_SIZE` in bits
  - `NUMERIC_STORAGE_SIZE` in bits

Access to Run Time Environment (2)

- New ordinary intrinsic functions: reading parameters from command line
- Get entire command line

```
CALL GET_COMMAND ([COMMAND,LENGTH,STATUS])
```

  - COMMAND (optional) character string
  - LENGTH (optional) number of characters of command

- Number of command line arguments

```
max_number = COMMAND_ARGUMENT_COUNT ()
```

- Get specific argument

```
CALL GET_COMMAND_ARGUMENT (NUMBER[,VALUE,LENGTH,STATUS])
```

  - character string VALUE returns argument NUMBER (0 <= .. <= max_number) with LENGTH single characters

Access to Run Time Environment (3)

- Intrinsic subroutine for getting environment variables

```
CALL GET_ENVIRONMENT_VARIABLE(NAME[,VALUE,LENGTH,STATUS,TRIM_NAME])
```

  - NAME is the environment variable
  - content is in VALUE (optional)
  - with LENGTH characters (optional)
  - STATUS indicates success or failure (optional)
  - TRIM_NAME=.true. if trailing blanks are considered as significant (optional)

```
PROGRAM test_getenv
 CHARACTER(len=255) :: homedir
 CALL get_environment_variable("HOME", homedir)
 WRITE (*,*) TRIM(homedir)
END PROGRAM
```

Access Control of `MODULE` Variables

- `protected` attribute (in addition to `public` and `private`: reference of a module variable without allowing it to be changed outside the defining module

```
MODULE temperature_module
  REAL,PROTECTED :: temperature_c = 0, temperature_f = 32
CONTAINS
  SUBROUTINE set_temperature_c(new_value_c)
  REAL,INTENT(IN) :: new_value_c
    temperature_c = new_value_c
    temperature_f = temperature_c*(9.0/5.0) + 32
  END SUBROUTINE
  SUBROUTINE set_temperature_f(new_value_f)
  REAL,INTENT(IN) :: new_value_f
    temperature_f = new_value_f
    temperature_c = (temperature_f - 32)*(5.0/9.0)
  END SUBROUTINE
END
```

MODULES: Extensions on the USE Statement

- use module-name, rename-list
  - Rename-list: local-name => original-name

- New: operators might have local names by renaming them in the USE statement

```
use my_module, operator(.myadd.) => operator(.add.)
```

  - .myadd. denotes .add. accessed by the module

MODULES: Extensions on the USE Statement (2)

- Use intrinsic and non_intrinsic to differentiate between build in and own modules
- Standard intrinsic modules:
  - IEEE modules
    - ieee_arithmetic,
    - ieee_exceptions,
    - ieee_features
  - ISO_FORTRAN_ENV intrinsic module
  - ISO_C_BINDING module

```
use, intrinsic     :: ieee_features
use, non_intrinsic :: my_module
```

Submodules

- Structuring a module into component parts, which may be in separate files
- Separating the definition of a module procedure into different parts
  - Interface, which is defined in the module
  - Body, which is defined in the submodule
- Entities are accessible by host association
- A submodule can contain entities of its own
- Submodules may have submodules
- No access by use association
- Submodules may be referenced recursively

Submodules

```
module points
 type :: point
  real :: x,y
 end type point
 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
```

Alternative declaration without restating the properties of the interface:

```
submodule (points) points_a
contains
 module procedure point_dist
  !NO interface variables here
  point_dist = sqrt((a%x-b%x)**2+ &
                    (a%y-b%y)**2)
 end procedure point_dist
end submodule points_a
```

Interface Block Extensions

- The module procedure statement in interface blocks has been changed
- Keyword `module` is now optional
- A procedure may appear in more than one interface block
- Allowed in an interface block are now
  - external procedures
  - dummy procedures
  - procedure pointers
  - module procedures

```
interface print
 module procedure print_my_type  !Fortran95
end interface
```

```
interface print
  procedure :: print_my_type  !Fortran 2003
end interface
```

Procedure Statement

- The procedure statement allows to declare a procedure
  - subroutine or function
  - implicit, explicit, intrinsic
  - by type or interface or abstract interface
  - within a derived type
- Allowed attributes
  - public, private
  - bind(c, [name=char_string])
  - intent
  - optional
  - pointer
  - save

```
procedure (my_subroutine) ::   new_subroutine
procedure (my_function)   ::   new_function
procedure(sin)            ::   my_sinus
procedure(real(kind=rk))  ::   real_function
! same as: real,external  ::   real_function
procedure()               ::   simple_external
! same as: external       :: simple_external
procedure(my_abstract_interface) ::   this_fun
```

Abstract Interfaces

- Abstract interfaces declare the interface of procedures without referencing any actual procedure
- Argument keyword names and their types, kinds, ranks are declared
- The name of the abstract interface may be used in a procedure(..) statement

```
abstract interface
 subroutine sub_interface()
 end subroutine sub_interface
 real (kind=8) function fun_interface()
 end function fun_interface
 subroutine example(var,arr)
  real:: var
  integer,dimension(:,:) :: arr
 end subroutine example
end interface
```

```
procedure(sub_interface) :: sub1, sub2
procedure(fun_interface) :: fun1, fun2
procedure(example) :: ex1, ex2  ! with 2 args
```

Declaration statements

Abstract Interfaces and Procedures

- Define an abstract interface to enforce procedures to have exactly that interface
- Will be important for procedure pointers

```
program main
 abstract interface
  subroutine sub_interface()
  end subroutine sub_interface
  real (kind=8) function fun_interface()
  end function fun_interface
  subroutine example(var,arr)
   real :: var
   integer, dimension(:,;) :: arr
  end subroutine example
 end interface
```

```
…
implicit none
  procedure (sub_interface) :: &
            mpi_init, mpi_finalize
  procedure (fun_interface) :: &
                           mpi_wtime
  real (kind=8) :: time

  call mpi_init
  time = mpi_wtime()
  call mpi_finalize
end program main
```

Module `ISO_C_BINDING`

- Named constants
  - kind parameter values for intrinsic types integer, real, complex, logical and character;
  - negative values indicate limited interoperability either no corresponding type is available or different precision or range
  - see sample program for list of all named constants
- Derived types
  - c_ptr and c_funptr for interoperability with C objects and function pointer types
  - named constants c_null_ptr and c_null_funptr for null values in C

```
use, intrinsic :: iso_c_binding
```

Intrinsic Types

Fortran Type	Corresponding C Type
integer (kind=c_int)	int
integer (kind=c_short)	short int
integer (kind=c_long)	long int
real (kind=c_float)	float
real (kind=c_double)	double
character (kind=c_char)	char
etc.	

Value Attribute

- “Call by value” is possible in Fortran!
- Restrictions:
  - No pointer, allocatable or procedure attribute
  - No intent in/inout, volatile attribute
  - Characters: length must be known at compile time

```
subroutine modify_arguments_f(cbr,cbv)
  integer (kind=4)           :: cbr
  integer (kind=4), value    :: cbv

  cbr = cbr + 1;
  cbv = cbv + 1;

end subroutine modify_arguments_f
```

cbr is altered on return
cbv is not altered

Value Attribute

- C Procedure:

```
void modify_arguments_c (int *cbr, int cbv)
{
  *cbr +=1; cbv +=1;
}
```

- Fortran:

```
interface
 subroutine modify_arguments_c(cbr,cbv) bind(c,name='modify_arguments_c')
  use, intrinsic :: iso_c_binding
  integer (kind=c_int)         :: cbr
  integer (kind=c_int), value :: cbv
 end subroutine modify_arguments_c
end interface
```

Invoking a Subroutine(!) from C in Fortran

- C Procedure:

```
void Copy (char in[], char out[]);
```

Fortran:

```
use, intrinsic :: iso_c_binding, only : c_char, c_null_char
interface
 subroutine copy(in, out) bind(c,name='Copy')
  use, intrinsic :: iso_c_binding, only: c_char
  character (kind=c_char), dimension(*) :: in, out
 end subroutine copy
end interface
character(len=10, kind=c_char) :: &
          digit_string = c_char_'123456789' // c_null_char
character(kind=c_char) :: digit_arr(10)
call copy(digit_string, digit_arr)
print '(1x, a1)', digit_arr(1:9)
end
```

Global Data

- Changes to variable in one language effects the corresponding variable in the other language.
- Save attribute is specified by default.
- Restrictions:
  - Only one fortran variable can operate with one C variable
  - `bind` not allowed for module procedure
  - `equivalence` statement is not applicable

```
module my_c_global_data
use, intrinsic :: iso_c_binding
integer(c_int), bind(c, name=„my_c_global_var“) :: my_c_var
…
end module my_c_global_data
```

Arrays

- Interoperable type and type parameter
- Fortran: more-dimensional arrays are stored columnwise
  C: more-dimensional arrays are stored rowwise
- Arrays of explicit size in Fortran and C

```
integer(c_int) :: fa(18,3:7), fb(18,3:7,4) < - > int ca[5 ][18], cb[4][5][18]
```

- Assumed size in Fortran array <--> C array is of unspecified size

```
integer(c_int) :: fa(18,*), fb(18,3:7,*) < - > int ca[ ][18], cb[][5][18]
```

C Pointer Type

- Derived types `c_ptr` and `c_funptr` are interoperable with C object and function pointer type.

- `C_f_pointer(cptr,fptr,shape)`
  associates a data pointer with target of a C pointer and specifes its shape.

- `C_f_procpointer(cptr,fptr)`
  associates a procedure pointer with the target of a C function pointer.

Example: C Pointer Type

```
module mod_grid
 use, intrinsic :: iso_c_binding
 implicit none
 type, bind(c)  :: grid
   integer (kind=c_int) :: nx
   type (c_ptr)     bbb :: x
 end type grid
contains
 subroutine spacing(g) bind(c,name='spacing')
  type (grid), intent(in) :: g

  integer (kind=4) :: i,j
  real (kind=c_float), pointer :: x(:)

  call c_f_pointer(g%x, x,(/g%nx/))
  …
 end subroutine spacing
end module mod_grid
```

Allocating Arrays in Fortran

- c_loc(x) and c_funcloc(x) return the C address of the argument with type c_ptr or c_funcptr respectively.

```
module mod_grid
…
real (kind=c_float), pointer :: x(:)
…
contains
 subroutine init_grid(nx,g) bind(c,name=„init_grid')
  integer (kind=c_int) :: nx
  type (grid), intent(in) :: g
  …
  g%nx = nx
  allocate(x(g%nx))
  g%x = c_loc(x)
  …
 end subroutine init_grid
end module mod_grid
```

Support for Usage of IEEE Arithmetic

- Standard IEEE 754 – 1985 for binary floating point arithmetic
  - Closed model for representations of numbers by -0, NaNs, ∞ and -∞
- Iso Standard 559 : 1989. binary floating-point standard for microprocessor systems
- Access to the different features by intrinsic modules
  - Ieee_features
  - Ieee_exceptions
  - Ieee_arithmetic
- Include private variables: existence of a variable indicates the feature

Module IEEE_FEATURES

- The module `ieee_features` defines named private constants corresponding to IEEE features
- Existence of a constant signals the compiler to provide appropriate code
- Code may be slower on some processors by the support of some features
- The effect is similar to that of a compiler directive (scoping unit)
- Level of support can be found out by the use of inquiry functions
- Accessibility implies support for at least one real type
- Example: In the scoping unit the compiler creates code supporting IEEE divide

```
use, intrinsic :: ieee_features, only: ieee_divide
```

Module IEEE_EXCEPTIONS

- Five exception flags (of derived type `ieee_flag_type`) are available:
  - `ieee_overflow` → (∞, -∞, HUGE(x), -HUGE(x))
  - `ieee_divide_by_zero` → (∞, -∞)
  - `ieee_invalid` (0*∞, 0/0, NaN*x)
  - `ieee_underflow` → `(TINY(x), 0)`
  - `ieee_inexact` → „Rounding errors“
- Function `ieee_support_flag (flag [,X])` returns .true., if the flag is supported on the processor (for reals of the same kind type parameter as the real argument `x`).
- Call of `ieee_get_flag (flag, flag_value)` returns in variable `flag_value` .true., if the corresponding exception flag is signaling
- An exception does not signal, if this could not arise during execution of code. The following piece of code must not signal divide by zero.

```
where (a > 0.0) a= 1.0/a
```

Module IEEE_ARITHMETIC

- Rounding mode is controlled by the non-elemental functions:
  - `ieee_get_rounding_mode(round_value)`
  - `ieee_set_rounding_mode(round_value)`
  - `round_value` is of derived type `ieee_round_type`

- A derived `type ieee_round_type` can have the following values:
  - `ieee_nearest` rounds the exact result to the nearest
  - `ieee_to_zero` rounds the exact result towards zero
  - `ieee_up` rounds the exact result towards +infinity
  - `ieee_down` rounds the exact result towards -infinity
  - `ieee_other` rounding mode does not conform to the IEEE Standard

Example for Rounding Mode

```
program main
   use, intrinsic :: ieee_arithmetic
   implicit none

   type (ieee_round_type) :: round_value
   …      ! Store the rounding mode
   call ieee_get_rounding_mode(round_value=round_value)
   call ieee_set_rounding_mode(round_value=ieee_up)
   …      ! Restore the rounding mode
   call ieee_set_rounding_mode(round_value=round_value)
 end program main
```