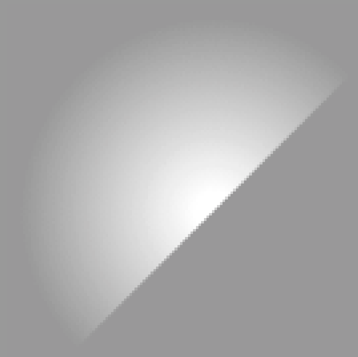

COMPLIMENTS • CON COMPLIMENTI • SALUDOS • MIT EMPFEHLUNG • AVEC COMPLIMENTS

Fortran 2000

Bill Long

16-May-2003

EXTREME PERFORMANCE! POWERED BY EXPERIENCE

Fortran 2000

Fortran 2000 is the next major revision of Fortran. Expect final version in 2004. This presentation is based on the April 29, 2003 03-007. THIS DRAFT MAY NOT MATCH THE FINAL DOCUMENT.

Major new features:

- C interoperability
- Object oriented programming
- IEEE support
- Asynchronous I/O

Presentation organization

- Basic Syntax
- Declarations
- Procedures
- Basic Operations
- I/O
- Future

Basic Syntax

Slide Notation

Code examples have colored text:

BLACK - Fortran 95 standard conforming
PURPLE - Fortran 2000 feature already implemented
BLUE - Fortran 2000 feature for early implementation
RED - Fortran 2000 feature for later implementation
ORANGE - C source code

Examples:

```
type, bind(c) :: struct
use, intrinsic :: ieee_exceptions
type, extends(foo) :: bar
```

Basic Syntax changes

Names up to 63 characters long.

Up to 255 continuation lines allowed.

Use [ ] as alternative to (/ /) for array constructors.

Named constants as parts of complex constants.

```
real,parameter :: zero = 0.0, one = 1.0
complex :: eye
eye = ( zero , one )
```

Printable ASCII characters now required in character set.

\ [ ] { } ` ^ | # @ ~

Declarations

Declarations

Module object access; protection

Protected attribute: `integer, protected :: ncpus`

Mixed component access:

```
type,private :: foo
   integer,public  :: bar1
   integer,private :: bar2
end type foo
```

Public entities of private type: `type(foo),public :: x`

Allocatable components

Allocatable components:

```
type :: foo
   real,allocatable :: bar(:)
end type foo
```

Contrast with f95:

```
type :: foo_old
   real,pointer :: bar(:)
end type foo_old
```

Allocatable character scalars

Allocatable character scalars are allowed.

```
character( len = : ),allocatable :: string

allocate( character(16) :: string)
```

Intrinsic Modules

Intrinsic Modules : supplied as part of compiler package

use,intrinsic :: iso_c_binding

use,intrinsic :: ieee_exceptions

use,intrinsic :: ieee_arithmetic

use,intrinsic :: ieee_features

use,intrinsic :: iso_fortran_env

ISO_C_BINDING module

The ISO_C_BINDING module contains definitions used for C interoperability. A subset includes KIND values:

c_signed_char, c_short, c_int, c_long, c_long_long
c_float, c_double, c_long_double
c_float_complex, c_double_complex, c_long_double_complex
c_char

Some character constants:

c_null_char, c_form_feed, c_new_line, c_carriage_return

New types:

c_ptr, c_funptr

ISO_FORTRAN_ENV module

The ISO_FORTRAN_ENV module contains constants that characterize the external environment.

I/O Units:

input_unit, output_unit, error_unit

I/O Status:

iostat_end, iostat_eor

Storage Unit sizes:

numeric_storage_size, character_storage_size, file_storage_size

Interoperation with C global objects

```
module global_data
   use,intrinsic :: iso_c_binding

   type,bind(c) :: flag_type
      integer(c_long) :: ioerror_num, fperror_num
   end type flag_type

   type(flag_type),bind(c) :: error_flags

end module global_data
```

```
typedef struct{ long ioerror_num, fperror_num; } flag_type

flag_type  error_flags;
```

Interoperation with C global objects

```
module global_data2
   use,intrinsic :: iso_c_binding
   integer(c_int),bind(c, name="FunnyCaps") :: funnycaps

   common /block/ r, s
   common /tblock/ t
   real(c_float) :: r,s,t
   bind(c) :: /block/, /tblock/

end module global_data2
```

```
int  FunnyCaps;
struct {float r, s} block;
float tblock;
```

Parameterized Derived Types

Parameterized Derived Types:

```
type :: tridiag (k, n)
   integer, kind     :: k        ! k must be known at compile time
   integer, length   :: n        ! n can to deferred to run time
   real(k)  :: upper(n-1)
   real(k)  :: diag(n)
   real(k)  :: lower(n-1)
end type tridiag

integer,parameter :: rk = selected_real_kind(12,100)
type(tridiag(8,20))                 :: mat20
type(tridiag(rk,:)),allocatable  :: mat(:)

allocate( type(tridiag(rk,20)) :: mat(4)) ! dynamic allocation
```

Extended types

Types that do not have the sequence or bind(c) attribute may be extended, implementing a single inheritance scheme for OOP.

```
type :: dna
   integer,allocatable :: ascii_text(:)
   integer             :: length
end type dna

type,extends(dna) :: ocdna
   integer :: ssdid, ssdsize, state
end type ocdna
```

The type ocdna has inherited components ascii_text, length, as well as a hidden component named dna of type(dna) that is just the parent components.

Volatile attribute

VOLATILE is an attribute that standardizes an existing extension.

```
integer, volatile :: flag
```

The memory associated with a volatile object may be modified by means not visible in the current program unit. The compiler must reload the value from memory for each use.

Initialization expressions

Many of the restrictions on initialization expressions are removed. In particular, references to most intrinsic functions are allowed.

```
real,parameter :: pi = acos(-1.)
```

Import statement

Interface blocks are local scoping units. IMPORT allows use of definitions from the host scoping unit.

```
type foo
   integer :: foo_int
end type foo

interface
   function bar(x) result(bar_res)
     import foo
     type(foo) :: x
     integer     :: bar_res
   end function bar
end interface
```

International character sets

Support is added for an extended character set for the values of character variables and constants.

Kind values:

selected_char_kind(NAME)
where NAME = “DEFAULT”, “ASCII”, or “ISO_10646”

ISO_10646 specifies the UCS-4 (32-bit) character set.

```
integer,parameter :: ucs4=selected_char_kind('iso_10646')
character(len=5,kind=ucs4) :: c

c = ucs4_"     "
```

Procedures

Procedures

Allocatable dummy arguments

Dummy arguments can be allocatable, allowing a procedure to allocate space for returned data arrays.

```
integer,allocatable  :: db(:)
call sub(db, nwords)

subroutine sub(db,n)
integer,allocatable :: db(:)
integer  :: n
read *, n
allocate(db(n))
read *, db
end subroutine sub
```

Allocatable function results

Allocatable function results are a variation on allocatable dummy arguments.

```
function foo(x) result(foor)
   real,dimension(:),intent(in)   :: x
   real,dimension(:),allocatable :: foor
   …
end function foo
```

Intent for pointer arguments

Intent specification for pointer dummy arguments is allowed. The intent applies to the association status of the pointer, not to the definition status of a target of the pointer.

```
subroutine sub(p,dat)
integer,pointer,intent(in) :: p(:)
integer,target  :: dat(10)

p = 1               ! OK.
allocate(p(20))   ! illegal - changes the target of p.
p => dat            ! illegal - changes the target of p.

end subroutine sub
```

Interoperating with C functions

Interface blocks interoperate with C function prototypes.

```
use,intrinsic :: iso_c_binding
interface
   function foo (prt, val),bind(c,name="Foo") result (bar)
   import :: c_int, c_long
   integer(c_int)            :: prt, bar
   integer(c_long),value :: val
   end function foo
end interface
integer(c_int) :: x,n ; integer(c_long) :: y
…
 n = foo(x,y)
```

```
int Foo( int *prt, long val);
```

Procedure statement

The PROCEDURE statement is an extension of the module procedure statement from f90, used to define a generic interface. The specific procedures do not have to be contained in a module. Only their interfaces must be available.

```
interface sgemm
   procedure  sgemm_44, sgemm_48, sgemm_84, sgemm_88
   procedure  cgemm_44, cgemm_48, cgemm_84, cgemm_88
end interface sgemm

interface dgemm
   procedure  sgemm_44, sgemm_48, sgemm_84, sgemm_88
   procedure  cgemm_44, cgemm_48, cgemm_84, cgemm_88
end interface dgemm
```

Procedure declaration statement

The PROCEDURE statement can declare names to be external procedures, identify an interface, and declare a procedure pointer.

```
abstract interface;  function fun_r (x)
                      real,intent(in) :: x
                      real            :: fun_r
                      end function fun_r;     end interface

procedure(fun_r) :: gamma, bessel

interface ; subroutine sub_r(x); real :: x
            end subroutine sub_r;          end interface

procedure(sub_r) :: sub
procedure(real)  :: psi   ! Equivalent to real,external :: psi
```

Procedure pointers

The PROCEDURE statement can be used to specify procedure pointers. Procedure pointers are allowed as components of derived types.

```
procedure(fun_r),pointer :: special_fun => null()

special_fun => gamma

type proc_ptr
      procedure(fun_r),pointer :: fun
end type proc_ptr

type(proc_ptr),dimension(10) :: special

ans = special(i)%fun(arg)
```

Type bound procedures

Procedures can be bound to a type, automatically carrying the interface along with each variable of that type. Procedures are declared with either the PROCEDURE, GENERIC, or FINAL statements.

```
type strange_int
  integer :: n
contains
  generic :: operator(+) => strange_int_add_oper
end type
```

The interface for strange_int_add_oper must be supplied either explicitly or by defining the function in the same module.

Polymorphic objects

The CLASS type specifier is used to declare polymorphic objects. These declarations must be for dummy arguments, or have the allocatable or pointer attribute.

```
function strange_int_add_oper (a,b) result (c)
   class(strange_int),intent(in) :: a,b
   type(strange_int)             :: c

   c%n = iand(a%n + b%n, 1)
end function strange_int_add_oper
```

The variables a and b are type compatible with actual arguments of type strange_int or any extension of strange_int.

```
class(*)  :: x ! X is type compatible with any type object.
```

Select Type construct

The SELECT TYPE construct allows alternate execution paths based on the actual type of a polymorphic object.

```
type,extends(strange_int) :: strange_int_m
   integer :: m
end type strange_int_m

select type(a)
type is (strange_int)
   c%n = iand(a%n + b%n, 1)
class is (strange_int)
   i = min(a%m, b%m)
   c%n = iand(a%n + b%n, 2**i - 1)
   c%m = i
end select
```

Finalizers

Finalizers are a special type bound procedure that is executed whenever an object of a derived type becomes undefined. This would include the initial state of an intent(out) dummy argument, or the state of an unsaved local variable at procedure exit.

```
type foo
    real,pointer,dimension(:) :: bar
contains
    final :: foo_cleanup
end type

subroutine foo_cleanup(x)
   class(foo) :: x
   deallocate(x%bar)
end subroutine foo_cleanup
```

New intrinsic functions

Optional KIND arguments added to many functions that return default integer results. Example: SIZE.

MAX and MIN allow character arguments.

EXTENDS_TYPE_OF and SAME_TYPE_AS, to inquire about extended types

NEW_LINE returns the value of the newline character, which is achar(10) on almost every system.

MOVE_ALLOC changes the address of an allocatable object.

C interoperability intrinsics

C_LOC(fort_arg) - returns a type(c_ptr) pointer to argument

C_ASSOCIATED(cp1 [, cp2]) - similar to associated, but for arguments of type(c_ptr)

C_F_POINTER - forms a Fortran pointer from a type(c_ptr)

C_FUNLOC(fort_proc) - returns a type(c_funptr) pointer to the Fortran procedure argument

C_F_PROCPOINTER - forms a Fortran procedure pointer from a C function pointer.

New environment intrinsics

COMMAND_ARGUMENT_COUNT,
GET_COMMAND, and GET_COMMAND_ARGUMENT
to get information about the command line.

GET_ENVIRONMENT_VARIABLE to get value of an environment variable.

IS_IOSTAT_END and IS_IOSTAT_EOR to determine
if an iostat value is an end of file or end of record indicator.

Basic Operations

Derived type constructors

Derived type constructors are extended to allow keywords an allocatable and procedure components.

```
type foo
   integer :: ii
   real,allocatable :: bar(:)
end type foo

type(foo) :: fobj

fobj = foo( ii = 1, bar = null() )
```

Enhanced array constructors

Allow type spec in an array constructor:

```
integer,parameter :: rk = selected_real_kind(12,100)
real(rk),dimension(4) :: spin
character(7) :: names(3)

spin = (/ real(rk) :: 0., 1., 0., 1.  /)

names = [ character(len=7) :: "Brian", "Jeff", "Melanie" ]
```

Assignment to allocatable variables

Allocatable components requires new rules for assignment. These are extended to ordinary allocatable objects as well.

```
type foo
   integer,allocatable,dimension(:) :: bar
end type foo

type(foo) :: f1,f2

allocate(f1%bar(100))
f1%bar(:) = 1

f2 = f1       ! f2%bar gets automatically allocated here
```

Assignment of allocatable variables

Same rules for allocatable intrinsic type variables.

```
 real,allocatable,dimension(:) :: a,b,c

allocate(a(10),b(20))
a = 1.2
b = 1.3
c = a          ! c allocated with size of 10
c = b          ! c reallocated with size of 20
c(:) = a(:)  ! illegal - array section sizes do not match
```

Assignment for characters

New assignment rules similar to allocatable arrays apply to allocatable character scalars as well.

```
character(len=:),allocatable :: string

allocate( character(16) :: string)
string = “0123456789abcdef”

string(:) = “pad”     ! padded with 13 blanks on right
string    = “short”   ! reallocated with len = 5
```

This new feature effectively provides a varying length string facility in Fortran.

Associate construct

The ASSOCIATE construct provides a shorthand for expressions and derived type objects that appear in statements.

```
do i = 1, genome(ng)%chr(nc)%dblen
   genome(ng)%chr(nc)%db(i) = iand(genome(ng)%chr(nc)%db(i),255)
end do
```

```
associate (x => genome(ng)%chr(nc) )
   do i = 1,x%dblen
      x%db(i) = iand(x%db(i), 255)
   end do
end associate
```

Lower bounds in pointer assignment

Lower bounds in pointer assignments can be specified.

```
real,pointer :: p(:)
real,target   :: t(100)

p => t(2:5)          !  p(1) has target of t(2)  - f95 rules

p(2: ) => t(2:5)  !  p(2) has target of t(2)  - new feature
```

Pointer rank remapping

Pointers of higher rank can have rank 1 targets through rank remapping. The rank 1 target may be more useful in some circumstances (as an argument to an old f77 function, for example) while the higher rank version may be more clear in computation expressions.

```
real,pointer :: p(:,:)
real,target   :: t(100)

p(1:10, 1:10) => t
```

Array reallocation

A new intrinsic is provided to simplify reallocation of an array.

```
integer,allocatable,dimension(:) :: x,tmp

allocate(x(20))
! …                                   ! (Old method)
allocate(tmp(40))                     ! allocate(tmp(20))
tmp(1:20) = x                         ! tmp = x
call MOVE_ALLOC ( tmp, x )            ! deallocate(x)
                                      ! allocate(x(40))
                                      ! x(1:20) = tmp
                                      ! deallocate(tmp)
```

IEEE features

The IEEE_FEATURES module MAY define these constants of type IEEE_FEATURES_TYPE:

ieee_datatype
ieee_nan
ieee_inf
ieee_denormal
ieee_rounding
ieee_sqrt
ieee_halting
ieee_inexact_flag
ieee_invalid_flag
ieee_underflow_flag

Undefined constants correspond to unsupportable features.

IEEE arithmetic control

The IEEE_ARITHMETIC intrinsic module defines these constants of type IEEE_CLASS_TYPE:

ieee_signaling_nan	ieee_quiet_nan
ieee_negative_inf	ieee_positive_inf
ieee_negative_normal	ieee_positive_normal
ieee_negative_denormal	ieee_positive_denormal
ieee_negative_zero	ieee_positive_zero
ieee_other_value	

and these constants of type IEEE_ROUND_TYPE:

ieee_nearest	ieee_to_zero
ieee_up	ieee_down
ieee_other	

IEEE arithmetic functions

The IEEE_ARITHMETIC module defines 28 functions to inquire about ieee floating point support and state.

Examples:

ieee_support_standard
ieee_support_inf

ieee_copy_sign
ieee_is_nan

ieee_get_rounding_mode
ieee_set_rounding_mode

IEEE exception control

IEEE_EXCEPTIONS intrinsic module defines:

types:

- ieee_flag_type
- ieee_status_type

values:

- ieee_overflow
- ieee_divide_by_zero
- ieee_invalid
- ieee_underflow
- ieee_inexact

routines:

- ieee_support_flag
- ieee_support_halting
- ieee_get_flag
- ieee_get_halting_mode
- ieee_get_status
- ieee_set_flag
- ieee_set_halting_mode
- ieee_set_status

I/O

I/O

Asynchronous I/O

Asynchronous I/O is supported with syntax to replace the old buffer in and buffer out statements. The “YES”/”NO” values need to be initialization expressions so they are known at compile time.

OPEN (UNIT=10, …. ASYNCHRONOUS = “YES” …)

READ (10, … ASYNCHRONOUS=“YES”, ID = idw, …)

WAIT (10, ID=idw)

Without an ID, the wait applies to all operations on the unit. CLOSE and INQUIRE have an implied wait.

Stream I/O

Steam access is a new alternative to Sequential and Direct access.

Both formatted and unformatted files can have stream access.

Formatted files have no record structure although embedded new_line characters may be used by the program.

Unformatted files to not have record length information embedded.

OPEN (… ACCESS = “STREAM” …)

A location within the file may be specified by the POS= specifier in the READ or WRITE statement.

FLUSH statement

The FLUSH statement provides a standard conforming syntax for the flush library routine. There are two forms:

```
FLUSH  10

FLUSH (UNIT = 10, IOSTAT = n)
```

Other allowed specifiers: IOMSG and ERR.

Some files do not support flush operations. In that case, the IOSTAT variable is set to a negative value.

On Cray systems, stdout automatically flushes.

Comma mode

The OPEN statement has a DECIMAL specifier for formatted I/O. If the value 4.3 is to be written to the file

DECIMAL = “POINT” -> 4.3 is output
DECIMAL = “COMMA” -> 4,3 is output

The default is “POINT”. This is an internationalization feature.

Note that using DECIMAL=“COMMA” disables the comma as a value separator in list-directed I/O. In that case, a semi-colon is used instead of comma as the value separator.

Rounding Modes

The OPEN statement supports a ROUND specifier to control numeric rounding of real values for formatted I/O. Allowed values are:

ROUND = "UP"
"DOWN"
"ZERO"
"NEAREST"
"COMPATIBLE"
"PROCESSOR_DEFINED"

The default value is processor dependent. On an IEEE machine, NEAREST is the IEEE meaning.

Text Encoding

The OPEN statement supports an ENCODING specifier that controls how the text in a formatted file is interpreted. The allowed values are:

ENCODING = “UTF-8”
“DEFAULT”

The default is DEFAULT, which is ASCII on most systems. The UTF-8 option is for Unicode text - the ISO_10646 set of characters.

I/O qualifiers in I/O statements

Many of the specifiers from OPEN statements for formatted files can be included in READ and WRITE statements. These override the values from the OPEN statement. The changeable modes are:

BLANK
DECIMAL
DELIM
PAD
ROUND
SIGN

```
read (unit=10,fmt=*,round="up")  x
```

I/O error messages

I/O statements are allowed to have the IOMSG specifier that is set to a printable error message if the statement resulted in a error, end-of-file, or end-of-record condition. The message is stored in the specified scalar default character variable.

```
character(132) :: msg

read (10, iomsg = msg ,iostat=n)  x
```

The intention is that these messages should be similar to the error messages printed

Normally this is used in conjunction with IOSTAT.

User defined type I/O control

Users can write subroutines to specify how derived type I/O is done. Up to 4 routines can be supplied for a type with these generic specifiers:

read(formatted)
write(formatted)
read(unformatted)
write(unformatted)

These are typically generic type bound procedures.

Formatted transfers use the DT edit descriptor.

DTIO example

Recall the previous example, now enhanced with a dtio specification

```
type :: dna
    integer,allocatable :: ascii_text(:)
    integer             :: length
contains
    generic :: write(formatted) => fw_dna
end type dna
type (dna) :: hs_chr20
```

In printing the dna string, you want to only print the text, not the length, so default derived type I/O would not work. You could write the individual components, but that is not in the OOP spirit.

```
write (10, "(dt)") hs_chr20
```

DTIO function example

For the previous example, the user needs to supply the I/O routine with a specific interface - this will be called by the library I/O routines as part of the write statement.

```
subroutine fw_dna(dtv, unit, iotype, vlist, iostat, iomsg)
  class(dna), intent(in)        :: dtv      ! hs_chr20
  integer,intent(in)            :: unit     ! 10
  character(*),intent(in)       :: iotype   ! "DT"
  integer,intent(in)            :: vlist    ! not used in this example
  integer,intent(out)           :: iostat
  character(*),intent(inout) :: iomsg

  ! write out the first dtv%length characters in dtv%ascii_text
  ! set iostat based on results of the write
  end subroutine fw_dna
```

Future plans and options

Future

Beyond Fortran 2000

Some broad ideas for future versions of Fortran:

Submodules

- Separate procedure interfaces and definitions
- Avoid compilation cascades

CAF

- Parallel constructs are important in today's environment
- Looking for a high performance solution

Typeless

- Better handling of BOZ constants and non-numeric data
- Simplified interfaces for some subprograms
- Standardize some common extensions

Standards structure

How Fortran gets made:

ISO -> WG5 -> J3

WG5 collects proposals and specifies the requirements for Fortran

J3 is delegated to actually write the document defining Fortran

Next WG5 meeting at the end of July.

Next J3 meeting in August, to produce the ballot draft standard.

Second ballot and final approval in 2004.

Request for comments

We welcome comments on

- The current draft standard (j3-fortran.org)
- Priorities for implementing the new F2000 features

Send comments to

longb@cray.com

CRAY

END • FIN • FINALE • FINE

EXTREME PERFORMANCE!

POWERED BY EXPERIENCE