Introduction to Advanced Fortran

A brief overview of the contents of this training course

- How do we teach?
  - An example session with “Operator overloading”

- A quick history of Fortran standards over the last decades
  - From FORTRAN66 to Fortran 2008

- Parallel processing aspects that often improve program performance: paving way to High Performance Computing
  - OpenMP available as an useful add-on for performance oriented applications (not part of Fortran standard)
  - Co-Array Fortran (CAF) is part of F2008 standard

A brief overview of this 4-day training course

- Advanced Fortran course mainly concentrates on F2003/F2008 aspects
  - some Fortran95 less well known features covered, too

- Our major subjects in this course are
  - Object Oriented Fortran (OOF) and Co-Array Fortran (CAF)
  - Additionally wide coverage of OpenMP + Fortran included

- Minor subjects cover
  - Interfacing with C—programs / libraries (ISO C bindings)
  - Developments with ALLOCATABLE and POINTER variables

Some training course practicalities

- Each lecture is about 45min – 1h
- Between lectures usually an exercise session, ca. 1h
- A training day ends at 4pm, except on the first day: 5pm
- Lunch breaks at noon, lectures/exercises continue at 1pm
- Prerequisites: good knowledge of Fortran and Unix/Linux
- During exercises we will be using CSC’s Cray XC30 (Sisu)
How do we teach?

- We will try to make potentially difficult things look relatively easy to learn and understand.
- We tend to skip items that we think have less significance in everyday modern Fortran programming.
- A typical lecture contains many smaller subjects, which often contain the syntactical point of view followed by simplified code snippets.
- An exercise session, that follows each lecture session, allow students to practice with the just learned subjects.

Operator overloading ...

- A practical implementation of operator overloading usually involve creating a Fortran MODULE – file where all the necessary components will be placed.
- A MODULE contains derived data TYPE definition of the associated elements and creation of appropriate INTERFACE block to enable compiler to map the references to the new operators.
- Operator overloading can optionally be implemented to cover several (intrinsic) data types, or a mixture of types.

An example session: “Operator overloading”

- To improve readability of Fortran source code it is possible to overload existing operators – or even create your own ones (not covered here).
- Operator overloading applies to the mathematical operations like ‘+’, ‘-’, ‘*’, ‘/’ and assignment ‘=’.
- Binary operators involve two operands – residing in the left and right hand sides (LHS & RHS) of the operator.
- Unary operator (e.g. minus A ) only has one operator (RHS).
- Typical operator overloading usages are found in the field of vector and matrix computation.

Overloading summation operator ‘+’ for vectors

- Whilst it is usually easier to code this directly without any extra code for implementing operator overloading, it is pedagogically a useful case to go through here.
- To start with we introduce our simple vector type:

```fortran
TYPE vector_t
  REAL, allocatable :: v(:)
END TYPE vector_t
```

- In a more generalized case this TYPE may contain many more elements than are shown here.
  - We may also want to sum over mixture of various REAL kinds and/or INTEGRERs, too.
The next step is to introduce a summation function, which does the actual addition, given two vectors.

- We need to introduce the summation OPERATOR itself and INTERFACE blocks to enable access.
- We also would like have flexible initialization of the vectors to be summed up – we take an opportunity to introduce an assignment OPERATOR, too.
- Our initial values are all INTEGERs, so we only need to map them to our vector TYPE.

All the necessary code fractions are now placed in a MODULE -file... and referenced from the user source code, e.g.:

```fortran
PROGRAM main
USE overload
TYPE(vector_t) :: x1, x2, out
x1 = [1, 2, 3];
x2 = [10, 20, 30];
out = x1 + x2
print '(a,3(1x,g0))','out= ',out
END PROGRAM main
! The output is:
out= 11.00000 22.00000 33.00000
```

A bit of Fortran history

- As many know the name “Fortran” stands for “Formula translating system”
- First version became standard by John Backus & IBM in 1957
- Well suited for numerical and scientific computing
- Emphasis often in HPC – High Performance Computing
- A language for in computationally intensive areas like
  - Numerical weather prediction,
  - Finite element analysis & computational fluid dynamics,
  - Computational physics & computational chemistry

Evolution of Fortran standards over the few decades

- Mid-60’s the most significant version was labeled as ANSI standard, called FORTRAN66 (“Fortran” in capital letters 😊)
- In 1978 the former “de-facto” FORTRAN77 standard was approved and quickly became popular, contained e.g.:
  - IMPLICIT statement
  - IF – THEN – ELSE IF – ELSE – ENDIF
  - CHARACTER data type
  - PARAMETER statement for specifying constants
  - Generic names for intrinsic functions
Evolution of Fortran standards...

An important extension to FORTRAN77 was release of “military standard” Fortran enhancements (also in 1978) by US DoD, adopted by most FORTRAN77 compilers
- IMPLICIT NONE
- DO – ENDDO
- INCLUDE statement
- Bit-manipulation functions

All these were eventually incorporated to the next major standard release – Fortran90

Fortran90

A major step in keeping Fortran programming alive was introduction of Fortran90 standard in ‘91 (ISO), ‘92 (ANSI)
- Free format source input (up to 132 characters per line)
- Dynamic memory handling via
  - ALLOCATE / DEALLOCATE for ALLOCATABLE –arrays
  - Automatic array sizing in procedures from arguments / modules

Marked a number of features obsolescent, but did not delete any features – yet, e.g.:
- Arithmetic IF and computed GOTO –statements
- Alternate RETURN, and use of H(ollerith) in FORMAT –statements

Fortran95

A minor revision in 1997 (ISO) some features taken from High Performance Fortran (HPF) specification, e.g.
- FORALL and nested WHERE clauses to improve vectorization
- User-defined PURE and ELEMENTAL procedures
- Automatic DEALLOCATE when ALLOCATABLE out of scope
- POINTER and TYPE components default initializations

Deleted some features previously marked as obsolescent
- Use of non-INTEGERs as DO-loop parameters
- H(ollerith) edit descriptor in FORMATS
- ASSIGN and assigned GOTO

Fortran 2003 (F2003)

A significant revision of the Fortran95 standard (in 2004)
- Parameterized derived TYPES
- Object-Oriented Programming (OOP), e.g.
  - Type extension, (single) inheritance, polymorphism, ...
- Procedure POINTERs
- Interoperability with C (and other) languages
- O/S interfacing : command line arguments & env. variables
- ALLOCATABLE improvements
- POINTER improvements
- I0 edit descriptor for INTEGERs in FORMAT statements
Fortran 2008 (F2008)

A minor revision of Fortran 2003 – in 2010
– Certain new features not fully supported by some compilers
New features include
– Co-arrays
– Sub-modules
– CONTIGUOUS –attribute
– BLOCK –construct
– newunit= in OPEN –statement
– DO CONCURRENT –construct
– G0 edit descriptor for REALs in FORMAT statements

Summary

The Fortran standard has evolved since its early days, late 1950s, and contains many modern language features, like Object Oriented Programming (OOP – with a twist “called” OOF) as well as Co-Array Fortran (CAF) to express (massive) parallelism.

OpenMP shared memory parallelism directives – though not part of the Fortran standard – enable performance oriented applications to gain benefits on multi-core computer architectures that are available everywhere today.

Interfacing with C-language is now standardized, so is the interface with operating system (command line arguments, obtaining environment variables and running Unix-commands from Fortran).

Developments in ALLOCATABLE and POINTER variables are also noticeable.
Useful features since Fortran90

- Interfacing the operating system
  - Accessing command line arguments and environment variables
  - Running operating system commands from Fortran
- Enhancements in use of ALLOCATABLE variables
  - Derived data type can have ALLOCATABLE components
  - Automatic DEALLOCATE when variable goes out of scope
  - ALLOCATABLEs as dummy arguments and function can return ALLOCATABLEs
  - Automatic resizing of ALLOCATABLE variables
  - ALLOCATABLE scalars: variable length CHARACTER strings
  - Two essential POINTER assignment improvements
- CONTIGUOUS attribute
- Asynchronous I/O
- I0 and G0 edit descriptors allow dynamic output FORMATTing

Command line arguments

- Parameters to a program are very often given to programs as command line arguments
  - Input file(s), modify program behavior, etc.
- Fortran 2003 has (finally!) a standardized method for reading command line arguments
  - No need to use extensions such as GETARG and IARGC
- The standard function calls are, respectively
  - get_command_argument and command_argument_count
- In addition, in order to access the whole command line, use
  - get_command

Access separate command line arguments

\[
\text{call } \text{get\_command\_argument} (\text{number}, \text{value}, \text{length}, \text{status})
\]
- number is of type integer and denotes which argument to get
- value is of type character string and contains the value of the requested argument on return. If the actual argument is too short or long it is padded with blanks or truncated, respectively (optional)
- length is of type integer and contains the length of the requested argument on return (optional)
- status is of type integer. On successful return status is 0, -1 if value was too short to contain actual argument and 1 if argument could not be returned (optional)

Get the number of command line arguments

integer :: command_argument_count
Command line arguments ...

- Access the whole command line
  
  `call get_command(command[,length][,status])`
  
  - `command` is of type character string and contains the value of the command line on return. If the actual command line is too short or long it is padded with blanks or truncated.
  - `length` is of type integer and contains the length of the command line on return or zero if the length cannot be determined (optional).
  - `status` is of type integer. On successful return `status` is 0, -1 if value was too short to contain actual argument and 1 if argument could not be returned (optional).

Command line arguments: example

```fortran
program commandline
  implicit none
  character(len=256) :: line
  integer :: i, iarg, stat, clen

  iarg = command_argument_count()
  do i=1,iarg
    call get_command_argument(i,line,clen,stat)
    write (*,'(I0,A,A)') i,' : ',line(1:clen)
  end do

  call get_command(line,clen,stat)
  write (*,'(A)') line(1:clen)
end program commandline
```

Environment variables

- Besides command line arguments, environment variables are a common way to modify program behaviour.
- Fortran 2003 has (finally!) a standardized method for accessing values of environment variables.
- In Fortran 77/90/95 accessing `getenv` from C standard library requires passing character strings from Fortran to C and back.
  - Without ISO-C bindings (only in Fortran 2003/2008), error prone and nonportable methods needed.

Environment variables

- Access a value of an environment variable
  
  `call get_environment_variable(name,value[,length][,status][,trim_name])`
  
  - `name` is of type character string and contains the name of the requested variable.
  - `value` is of type character string and contains the value of the requested variable. If the actual variable value is too short or long it is padded with blanks or truncated. If the variable has no value or does not exist, `value` is set to blanks.
  - `length` is of type integer and contains the length of the requested variable on return if the variable exists and has a value and zero otherwise (optional).
  - `status` is type integer. If requested variable does not exist status is 1. If `value` was too short status is -1 and zero otherwise. For other return codes, see docs (optional).
  - `trim_name` is of type logical and sets if trailing blanks are allowed in variable names or not (optional).
Environment variables: example

```fortran
program environment
implicit none
character(len=256) :: enval
integer :: len, stat

call get_environment_variable('HOSTNAME', enval, len, stat)
if (stat == 0) write (*,'(A,A)') 'Host=', enval(1:len)
call get_environment_variable('USER', enval, len, stat)
if (stat == 0) write (*,'(A,A)') 'User=', enval(1:len)
end program environment
```

Executing commands

- Invoking external programs from within a program is occasionally needed
  - No source nor library API available for a useful program
- Fortran 2008 has (finally!) a standardized method for invoking an external command
- In Fortran 77/90/95 accessing `system` from C standard library requires passing character strings from Fortran to C and back
  - With Fortran 2003, ISO-C bindings can be used

Executing commands: example

```fortran
program execcommand
implicit none
integer :: estat, cstat

call execute_command_line('ls -al', .TRUE., estat, cstat)
if (estat==0) write (*,'(A)') 'Command completed successfully'
end program execcommand
```

- Execute a command line
  - `call execute_command_line(command[,wait][,exitstat][,cmdstat][,cmdmsg])`
    - `command` is a character string containing the command to be invoked
    - `wait` is logical value indicating if command termination is to be waited (.true., the default) or if the command is to be executed asynchronously (.false.) (optional)
    - `exitstat` is an integer value containing the return value of the command if `wait=.true.` (optional)
    - `cmdstat` is an integer value. It is assigned a value of zero if `command` executed successfully. For other return codes, see docs (optional)
    - `cmdmsg` is a character string containing explanatory message for positive values of `cmdstat` (optional)
Enhancements to ALLOCATABLE –variables

There are plenty of new features to dynamic memory allocation through ALLOCATABLE –variables since introduction of Fortran90 standard in early 1990s.

The two most common, which may already have used, perhaps without notice, are:
- Automatic DEALLOCATE when ALLOCATABLE goes out of scope
- Use of ALLOCATABLE components in derived data TYPEs

But there are more, less well known cases:
- We will start from the most common ones

Fortran90 standard introduced user defined data types, but dynamic components were restricted to POINTERs only.

This oddity was removed in Fortran 2003.

Allocation remains intact even after going out of scope, if TYPE was defined in a MODULE.

An ALLOCATABLE variable, which is declared as a local variable in procedure and gets ALLOCATEd, will be automatically DEALLOCATEd upon returning back from routine.

This is out of scope –rule holds as long as variables don’t have SAVE –attribute as well.

Automatic DEALLOCATE

An ALLOCATABLE variable can now appear in the procedure argument list.

Enables dynamic memory allocation in the callee based on sizing information.

Allocated (and perhaps initialized) data is returned to the caller.

The caller must remember to DEALLOCATE.

ALLOCATABLE as derived data TYPE component

Fortran90 standard introduced user defined data types, but dynamic components were restricted to POINTERs only.

This oddity was removed in Fortran 2003.

Allocation remains intact even after going out of scope, if TYPE was defined in a MODULE.

It’s not obvious for the caller to detect when ALLOCATEd data is no longer used by the callee.

The compiler can be instructed not to automatically deallocate ALLOCATEd data when it is no longer actually referenced.

To avoid automatic deallocation:

- Allocate the ALLOCATABLE variable in a module.

Fortran90 standard introduced user defined data types, but dynamic components were restricted to POINTERs only.

This oddity was removed in Fortran 2003.

Allocation remains intact even after going out of scope, if TYPE was defined in a MODULE.

An ALLOCATABLE variable can now appear in the procedure argument list.

Enables dynamic memory allocation in the callee based on sizing information.

Allocated (and perhaps initialized) data is returned to the caller.

The caller must remember to DEALLOCATE.

ALLOCATABLE as dummy argument

An ALLOCATABLE variable can now appear in the procedure argument list.

Enables dynamic memory allocation in the callee based on sizing information.

Allocated (and perhaps initialized) data is returned to the caller.

The caller must remember to DEALLOCATE.
**Function return value as ALLOCATABLE**

A function can now have ALLOCATABLE –attribute  
Works similarly as when ALLOCATABLE is a dummy argument  
Particularly useful in returning dynamically sized array as a result of CONTAINS –function value evaluation

**Allocation to an ALLOCATABLE variable**

Given an ALLOCATABLE array, which has not yet been explicitly allocated  
Assigning values to it via data or from another array triggers automatic allocation  
If array was already allocated, but its size was not sufficient, the extent size will be allocated automatically  
Use of this feature currently requires additional compiler flags, like `-assume realloc_lhs` (Intel) and `-ew` flag with Cray compiler

**ALLOCATABLE scalar**

These are especially useful with Fortran CHARACTER strings, whose length is decided at run time  
Allocation can either explicit or automatic (as per previous page)  
Again, requires additional compiler flags due to implicit assignments of ALLOCATABLEs

**Transferring allocation : MOVE_ALLOC**

Provides ALLOCATABLE equivalent of POINTER assignment: moves allocation to another memory location  
As result previous allocation becomes unallocated and new allocation holds previous data  
Arrays to (=A) and from (=B) must have the same type and rank
Two improvements to the POINTER assignments

The first improvement is that it now possible to set the desired lower bounds of a POINTER to any value, e.g.:

```fortran
REAL, target :: state_budget ( 1917 : 2013 )
REAL, pointer, dimension() :: A, B, C
A => state_budget
B => state_budget ( 1939 : 1945 )
C (1939:) => state_budget ( 1939 : 1945 ) ! ! New
```

The bounds of A are (1917,2013), but B has just (1,7)
The new LHS pointer, however, has bounds (1939,1945)

The 2\textsuperscript{nd} improvement is such that the TARGET of a multi-dimensional array POINTER can be a one-dim. array, e.g.:

```fortran
REAL, pointer :: memory(:), Matrix(:,:), Diagonal(:)
ALLOCATE(memory(N * N)) ! Space for N-by-N matrix
Matrix (1:N, 1:N) => memory  !! New stuff
Diagonal => memory(:N+1) ! ... or: memory(1:N*N:N+1)
```

After this the POINTER-array Matrix operates across all the memory region that has been allocated
Diagonal POINTER refers to the diagonal elements

CONTIGUOUS attribute

Depending on the hardware architecture, a unit stride data access (contiguous) can be two to three times faster than a constant non-unit stride (non-contiguous)

Typical optimization: use structure of arrays in stead of arrays of structures

In Fortran, having non-contiguous data is possible due to non-unit stride array indexing, for example:

```fortran
vector(::3) ! Every third element
```

In Fortran 2008, an assumed shape or pointer variable can be declared **contiguous**

- **CONTIGUOUS** type attribute, for example
  ```fortran
  real, contiguous :: x(:)
  real, pointer, contiguous :: y(:,:)
  ```

- **CONTIGUOUS** statement
  ```fortran
  contiguous [::] object-name-list
  ```

Testing contiguity

```fortran
logical :: is_contiguous(arr)
where arr is an array of any type. The function returns .true. if arr is contiguous and .false. otherwise.
```
CONTIGUOUS attribute ...

Potential performance benefits
- Array traversal and element address computation is simplified
- Better vectorization properties
- No need to generate auto-dispatch code for contiguous data separately

Simply contiguous: no need to declare the array as contiguous. Some examples:
- A whole array of that is not of assumed shape or pointer
- Continuous section of a (simply) contiguous array

Asynchronous I/O

Both input and output I/O can be asynchronous i.e.
- Other statements may be executed whilst I/O is in progress in the background

Caveat: it is an implementation dependent factor whether I/O actually is performed asynchronously

Asynchronous I/O ...

When all I/O statements are performed in the same routine, then the arrays / variables need not to be declared with asynchronous –attribute
- Only in OPEN/READ/WRITE

You must also not access array A whilst it is being processed by async I/O
It is a good practice to use identifier id for each asynchronous READ/WRITE and pass it to the WAIT statement
If you OPEN your file for asynchronous I/O, but perform actual operations (e.g. WRITE) in other procedure, then the arrays involved in the I/O must also be declared with asynchronous attribute.

```fortran
PROGRAM async_io
    INTEGER :: id, IA(1000000)
    CALL initialize(IA,size(IA))
    OPEN (10, ..., asynchronous='yes' )
    CALL async_write(10,id,IA,size(IA))
    CALL do_something_else_here()
    WAIT (10, id=id)
END PROGRAM async_io

SUBROUTINE async_write(iu, id, ia, n)
    INTEGER, intent(in) :: iu, n
    INTEGER, intent(in), asynchronous :: ia(n)
    INTEGER, intent(out) :: id
    WRITE(iu, id=id, asynchronous='yes' ) ia
END SUBROUTINE async_write
```

Non-blocking asynchronous I/O

By calling WAIT –function, you may potentially wait for a very long time before your I/O operation(s) are complete. Instead, you can also periodically call the INQUIRE –function and request the status of your pending asynchronous I/O operation(s) – and in the meantime do something else.

```fortran
LOGICAL :: status
INQUIRE (unit = iu, id = myid, pending = status)
```

If status==.FALSE. → outstanding I/O with myid not yet complete

Unfortunately INQUIRE (... , pending = ...) appears to be blocking rather than non-blocking even with id= parameter given : thus equals to WAIT

I0 and G0 edit descriptors in FORMAT

Dynamic sizing of REAL and INTEGER valued output

- I0 appeared in F2003 and G0 was introduced in F2008
- Correspond to C-language %d and %g fprintf –formats

Output fields become left justified with all the unnecessary leading blanks (and precision for REALs) removed

```
    INTEGER :: I = 12345
    REAL (kind=4) :: SP = 1.23e0
    REAL (kind=8) :: DP = 1.234567890d0
    WRITE(*,fmt=’("<I=",I0"," REALs=",2(G0,1X),"">")’) I,SP,DP
```

Output is (the “˽” denotes a space character)

```
<I=12345,˽REALs=1.230000˽1.234567890000000˽>
```

Summary

It is now possible to obtain command line arguments, access environment variables and run operating system commands in a standardized way

- Dynamic memory allocation has also been improved by means of enhancements to ALLOCATABLE variables
- Two new POINTERs improvements were introduced
- CONTIGUOUS –attribute for arrays enables more optimization
- Asynchronous I/O enables overlap with computation
- New edit descriptors enable dynamic sizing of output
For large applications, using only intrinsic types is often insufficient.

To have the data as larger objects, it is beneficial to group the data together as larger objects:

- Code becomes easier to read and maintain
- Cleaner interfaces
- Encapsulation of data

**Derived types**

```fortran
type person
  integer :: age
  character(len=20) :: name
end type person

type(person_type) :: p1
  integer :: p1_age
  character(len=20) :: p1_name

p1_age = 20
p1_name = 'John'
p1 = person(p1_age, p1_name)
call print_person_bad(p1_age,p1_name)
call print_person_good(p1)
```

**Derived types: declaration**

Derived type declaration in Fortran 90/95:

```fortran
type [[,access]: ] type_name
  [private]
  def_stmt
  [def_stmt]
end type [type_name]
```

- `access` field can be `public` or `private`
- Inside the type, `private` defines private component access
**Derived types: declaration**

- Derived type fields defined by `def_stmt`:
  - `type [[: attr_list:::] decl_list`
- `attr_list` is composed of `pointer` or `allocatable` and `dimension` attributes
- `decl_list` is composed of:
  - `name [(bounds)][:*char_len] [init]`
- In case of `pointer` attributes, `type` can point to enclosing type
- `bounds` is: for `allocatable` and `pointer` components and `[lower:]upper` for all others

**Derived types: examples of declaration**

```
type base_type
  integer :: field = 1
end type base_type

type :: array_type
  integer, pointer :: field1(:)
  ! Equivalently integer, dimension(:), pointer :: field1
  integer :: field2(10)
end type array_type

type recursive_type
  type(recursive_type), pointer :: next
end type recursive_type
```

**Derived types: component visibility**

- Restricting component access enforces good programming practice
  -Clearer division of responsibilities (who modifies what)
  -Better encapsulation
- Recommendation: restrict access to components of derived types whenever possible

```
type person_exposed
  integer :: age
  character(len=20) :: name
end type person_exposed

type person_encapsulated
  private
  integer :: age
  character(len=20) :: name
end type person_encapsulated

! In other module
type(person_exposed) :: p1
type(person_encapsulated) :: p2
! Exposed components
p1 % age = p1 % age + 1
! Encapsulated components
call increment_age(p2)
```

- A derived type is fully visible within the host module
- For other modules (after `use` or `import` -statement)
  - `public` types are fully visible
  - components of `public` types with `private` specifier are hidden
  - All `private` types are hidden
There is often need to change the kind parameter or the length of data
– Solution: create a “template” where the data fits allowing code re-use and better compiler optimizations
Idea similar to C++ templates

Fortran 2003 allows parameterization of
– Intrinsic types with KIND with and LEN attributes
– Derived types with integer KIND and LEN attributes

Parameterized type name definition becomes:
```fortran
type_name (type_spec_list)
```

List of type specifiers are of the form type_spec:
```fortran
[type_keyword =] type_param_value
```
The definitions for keyword = clause are consistent with those of intrinsic types: any further type parameter specifications must include a keyword

Default values for type parameters with default initialization syntax

Parameterized type name definition becomes:
```fortran
type_name (type_spec_list)
```

List of type specifiers are of the form type_spec:
```fortran
[type_keyword =] type_param_value
```
The definitions for keyword = clause are consistent with those of intrinsic types: any further type parameter specifications must include a keyword

Default values for type parameters with default initialization syntax

Type parameters
– Kind parameter KIND must be known compile time
– Length parameter LEN can be deferred until runtime

Type enquiry
– Variable: via implicitly declared LEN and KIND components
– Derived type: via LEN and KIND –components of the type

Parameterized type name definition becomes:
```fortran
type_name (type_spec_list)
```

List of type specifiers are of the form type_spec:
```fortran
[type_keyword =] type_param_value
```
The definitions for keyword = clause are consistent with those of intrinsic types: any further type parameter specifications must include a keyword

Default values for type parameters with default initialization syntax

Parameterized type name definition becomes:
```fortran
type_name (type_spec_list)
```

List of type specifiers are of the form type_spec:
```fortran
[type_keyword =] type_param_value
```
The definitions for keyword = clause are consistent with those of intrinsic types: any further type parameter specifications must include a keyword

Default values for type parameters with default initialization syntax

Parameterized type name definition becomes:
```fortran
type_name (type_spec_list)
```

List of type specifiers are of the form type_spec:
```fortran
[type_keyword =] type_param_value
```
The definitions for keyword = clause are consistent with those of intrinsic types: any further type parameter specifications must include a keyword

Default values for type parameters with default initialization syntax
Derived types: allocatable components

- Fortran 2003 allows derived types to have components with the ALLOCATABLE attribute
  - Derived type fields defined by `def_stmt`
    ```fortran
    type [[, attr_list][:] decl_list],
    where attr_list is composed of pointer, allocatable, dimension and contiguous (F08) attributes
    ```
  - Enhancements of F03 allocatable semantics apply to allocatable components of derived types
  - Recursive reference to enclosing type is allowed in F08

- In structure constructors for derived types
  - Pointer components: must be matched with target, pointer or `null()`
  - Allocatable components: must be matched with data item or `null()`
  - Initial state for allocatable components is unallocated

- Assignment semantics differ by component type
  - Pointer components: shallow copy, i.e., pointer assignment is performed
  - Allocatable components: deep copy, i.e., the required storage is automatically allocated and data is copied
Derived types: assignment semantics

```fortran
! Compute y = x + 1
some % nested % type % y = &
  some % nested % type % x + 1
associate(x=>some % nested % type %x,
  y=>some % nested % type %y)
! Compute y = x + 1
  y = x + 1
end associate
```

ASSOCIATE construct

- `ASSOCIATE` construct allows defining aliases to lengthy descriptions of variables for the duration of the associate block.

- `associate (association_list)
  block
der
end associate`

- List of associations is of the form association:
  `associate_name => variable`
  or
  `associate_name => expression`

Deeply nested types have a drawback of making the code more difficult to understand

- Not just a code readability problem: the compiler may not be able to perform certain optimizations when nested types are used

Within the block, `associate_name` takes the type, type parameters and rank from the association

- Inherited: polymorphism, type, array rank and shape. Also asynchronous, target and volatile attributes
- Not inherited: pointer, allocatable and optional attributes

If `associate_name` is a variable, it can be used as a variable within the block and is assignable

If `associate_name` is an expression, association is used for the expression value within the block and is not assignable
**Associate construct example**

```fortran
associate( x => some % nested % type % x,  
        y => some % nested % type % y,  
      z => some % nested % type % z)
! Compute z = sqrt(x**2+y**2)
    z = sqrt(x**2+y**2)
! Possibly other computations involving x, y and z
end associate
```

**Abstract Interfaces and Pointers**

Programs may have to call routines outside the program itself (external libraries) or provide services to others APIs. Wrong calling arguments may lead to errors during the executable linking phase or even when the executable is being run.

For external procedures, interfaces determine the type and properties of arguments and return values.

Defined by `INTERFACE` block:

```
interface
    subroutine not_dangerous(a, b, c)
        integer :: a, b, c
    end subroutine not_dangerous
end interface
```

The `interface-body` matches the subprogram header:
- position, rank and type of arguments
- return value type and rank (for functions)

Interface needed when procedures used as arguments.

```
integer :: x, y, z
x=1; y=1; z=1
! Call external subroutine without an interface
call dangerous(x,y,z)
! Call external subroutine with an interface
call not_dangerous(x,y,z)
```
Interfaces example

! LU decomposition from LAPACK
interface
  subroutine DGETRF(M, N, A, LDA, IPIV, INFO)
  integer :: INFO, LDA, M, N
  integer :: IPIV( * )
  double precision :: A( LDA, * )
  end subroutine DGETRF
end interface

! Euclidean norm from BLAS
interface
  function DNRM2( N, X, INCX ) result(double precision)
  integer :: N, INCX
  double precision :: X( * )
  end function DNRM2
end interface

Abstract interfaces

- Different procedures with a similar interface must be declared separately, leading to repeated code
- Declaration of a generic interface:
  abstract interface
  abstract-interface-body
  end interface
- Routines declared in abstract-interface-body must not have an actual implementation in the current scope

PROCEDURE statement

- Abstract interface can be referenced with procedure statement
  procedure ([iface]) [[, spec ...]] : decl-list
  where spec is one of:
  public, private, bind, intent, optional, pointer, save
  and list of declarations decl:
  name [=> null-init]
- null-init references intrinsic function null() with no arguments and may only appear with pointer
- Implicit interfaces similarly to external statement

Abstract interface example

! Abstract interface definition
abstract interface
  subroutine sub_with_no_args()
  end subroutine sub_with_no_args
  function trig(x) result(y)
  real, intent(in) :: x
  real :: y
  end function trig
end interface

! Procedure definition
procedure(sub_with_no_args) :: sub1
procedure(trig) :: mysin, mycos

! Implicit interfaces
! No known return value nor arguments
procedure() :: x
! Real return value, arguments unknown
procedure(real) :: y
Procedure pointers

- Procedure statement allows definition of pointers to procedures (explicit or implicit interface)
- A derived type can have procedure pointers as components
- When a procedure pointer is invoked as a component of a derived type, the object itself is passed to the procedure as the first actual argument
  - **pass** argument allows passing invoking object as any dummy argument
  - **nopass** argument allows not passing invoking object at all

Procedure pointers example

```fortran
! Interface definition
abstract interface
  function trig(x) result(y)
    real, intent(in) :: x
    real :: y
  end function trig
end interface

procedure(trig), pointer :: p1
p1 => mysin
write (*) p1(x) ! Prints mysin(x)

type base
  procedure(sub), pointer, pass(x) :: p
end type base
abstract interface
  subroutine sub(a,b,x)
    import base
    real :: a,b
class(base) :: x
  end subroutine sub
end interface

type(base) :: t1
t1 % p => mysub
! Equivalent to call t1 % p(1.0,2.0,t1)
call t1 % p(1.0,2.0)
```

Summary

- Derived types are a natural way to represent data
- Encapsulation of derived type components is possible with the PRIVATE attribute
- Parameterization of derived types is possible with KIND and LEN attributes
- Abstract interfaces and PROCEDURE statement provide type-safe access to external routines
- Procedure pointers provide functionality similar to C-function pointers and can be bound to derived types
What is object-oriented programming?

- Program is separated into interacting objects
- Objects couple the data and the methods operating on the data
- Generic programming: the actual type of data and the associated implementation may be encapsulated and abstracted
- Maintainability, readability and modifiability of the code are improved
- Performance penalty of a well-designed object oriented code is usually negligible

Class diagrams

In OO programming, UML (Unified Modeling Language) class diagrams are typically used to describe relationships between classes

- Used for conceptual and data modelling of the application structure
- Automatic generation of code from class diagrams possible (with Java and C++)

Class diagrams: class members

- **Name** describes class name
- Description of fields and methods visibility, type and arguments
- **Visibility specifiers**
  - "+" public
  - "#" protected
  - ":" private
  - ":_" static

<table>
<thead>
<tr>
<th>Name</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ field1</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>field2</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- field3</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_field4</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ method1(arg: type)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>method2(arg: type)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- method3(arg: type)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_method4(arg: type)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Class diagrams: instance relationships

- **Association**
  - "relates to", link between two or more classes

- **Aggregation**
  - "has a", part-whole or part-of relationship between two classes

- **Composition**
  - "owns a", a lifecycle aggregation between two classes

Class diagrams: class relationships

- **Generalization**
  - "is a", parent class generalizes the extending class; inheritance

- **Realization**
  - "uses", a use relationship between two or more classes

- **Multiplicity**
  - "count", countability of the relationship

Object-oriented programming in Fortran

- Fortran 2003/2008 supports object-oriented (OO) programming
  - Type extensions, polymorphism (single inheritance)
  - Type and object-bound procedures, finalizers, type-bound generics
  - Abstract interfaces and types

- Object model designed to maintain backwards compatibility with Fortran 95
**IMPORT statement**

- Interface does not access its environment via host association, i.e., **named constants** and **derived types** from **enclosing module not available**
  - USE statement requires breaking of encapsulation
- Fortran 2003 addresses this with import statement: `import [ ::] import-name-list`
  where **import-name** is an entity to be accessed from the host
- **import** without arguments makes **all** host entities accessible

**Type extension**

- Type extensions are used to extend the functionality of an existing type
- Type extensions are backwards compatible, because every instance of **extended** is also an instance of **base**

**Type extension**

- To be eligible for type extension, parent type must be **extensible**, i.e., be a derived type with neither **sequence** or **bind(C)** attributes
- Extensions can have zero additional components
- New type components or type parameters cannot have the same name as inherited ones
- Overriding of old type-bound procedures is allowed

**Type extension: EXTENDS**

- Type definition statement with **extends** attribute: `type [[,access][,extends(pname)] :: ] tname ...
  end type [tname]`
  - **pname** defines an existing derived type to be extended
  - All components as well as type parameters of the **parent type** **pname** are inherited by **extended type** **tname**
  - Type **tname** has an additional parent component of the derived type **pname** with the name **pname**
Type extension example

! Type describing a person
type :: person
  character(len=10) :: name
  integer :: age
end type person

! Employee has fields:
! name, age, salary, person
type, extends(person) :: employee
  integer :: salary
end type employee

! Staff has fields:
! name, age, salary, employee
type, extends(employee) :: staff
end type staff

Polymorphism

Polymorphism = “The provision of a single interface to entities of different types”, Bjarne Stroustrup, creator of C++

Object polymorphism: a variable can have instances from different classes as long as they are related by a common base class

Polymorphism: CLASS

- Type of a polymorphic variable may vary at run time
- Declaration of a polymorphic variable:
  class(type_name)[, class_attr ] :: name
  where class_attr may be omitted for dummy data objects, and is either pointer or allocatable otherwise
- Polymorphic variables are type-compatible with type_name and all extensions of type_name

Polymorphism limitations

- Polymorphic variable can only appear in an input/output list if it is processed by derived type output
- Intrinsic assignment to polymorphic variable is not allowed unless it is associated with a non-polymorphic variable (Fortran 2008 allows assignment of allocatable variables)
- Arrays of polymorphic variables are always homogeneous
  - Use an array of derived type having polymorphic pointer or allocatable component
Polymorphism example

```fortran
subroutine write_name(this)
  class(person) :: this
  write (*,*) this % name
end subroutine write_name

type(person) :: p1
type(staff), target :: p2
class(employee), pointer :: p3

p1 = person(name='Joe',age=20)
p2 = staff(name='Mike',age=42,salary=2000)
p3 => p2

call write_name(p1) ! Prints 'Joe'
call write_name(p3) ! Prints 'Mike'
```

Polymorphic object initialization

- Actual type of a polymorphic object is only known during runtime
- Polymorphic objects can be allocated only dynamically
- Polymorphic object allocation may be based on previously allocated type-compatible data

Polymorphic object initialization

- ALLOCATE statement extended to allow definition of type, type parameter values, type values and size (for array objects)
- Fortran 2003/2008 ALLOCATE statement
  ```fortran
  allocate([type_spec::] alloc_list [,stat=astat])
  ```
  where `type_spec` is `type_name` followed by type parameter values in parentheses

Sourced object initialization

- Values may be initialized based on a SOURCE object
  (single object in Fortran 2003, multiple objects in Fortran 2008)
  ```fortran
  allocate(alloc [source=expr] [,stat=astat])
  ```
- SOURCE object expression `expr` is type-compatible with `alloc`. For arrays, `expr` and `alloc` must be of the same rank
- In Fortran 2008 omitting array bounds from both `alloc` and `source` expressions is allowed
Sourced object initialization example

integer, parameter :: n = 5
integer :: i
real :: a(n)
real, allocatable :: b(:), c(:)
a = (/ (i, i=1,n) /)
allocate(b(lbound(a,1):ubound(a,1)), source=a) ! Fortran 2003
allocate(c, source=b) ! Fortran 2008

! a is [1, 2, 3, 4, 5]
! b is [1, 2, 3, 4, 5]
! c is [1, 2, 3, 4, 5]

Molded object initialization

- Fortran 2008 allows objects to have the shape, type and
type parameters without copying the object values by using MOLD
allocate(aloc [mold=expr] [,stat=astat])

- MOLD object expression expr must be type-compatible
with alloc. For arrays, expr and alloc must be of the
same rank

- After the allocation any default initialization values are
applied to alloc

Molded object initialization example

type(person) :: p1
type(employee) :: p2
class(person), allocatable :: p3, p4
p1 = person('Joe', 10)
p2 = employee('Mike', 42, 2000)
allocate(p3, source=p1) ! p3 is a copy of p1
allocate(p4, mold=p2) ! p4 has the same type as p2
call write_name(p1) ! Prints 'Joe'
call write_name(p2) ! Prints 'Mike'
call write_name(p3) ! Prints 'Joe'
call write_name(p4) ! Prints ' '

SELECT TYPE construct

- SELECT TYPE –construct allows the programmer to
determine the type of a
type dynamically during runtime

- Type determination is
relatively expensive –
avoid using in
performance critical parts
of the program
**SELECT TYPE construct**

- Type of polymorphic objects may be determined at runtime by using **SELECT TYPE** construct
  ```
  [name:] select type ([assoc_name =>] selector)
  [guard-stmt [name]]
  block
  ...
  end select [name]
  ```
- The **selector** may be named with **assoc_name** similarly to **ASSOCIATE** construct

**SELECT TYPE construct cont.**

- **SELECT TYPE** matches the block in the following way:
  1. If derived type and type parameters match exactly, the **block** is executed
  2. If derived type class and type parameters match, the **block** is executed. If more than one block matches, the **block** which is an extension of the type of all the others is executed (most closely matching type)
  3. The **block** following **class default** is executed
- Within the matched block, **selector** type is **type_spec**

**SELECT TYPE example**

```fortran
subroutine write_person(this)
class(person), intent(in) :: this
select type (this)
type is (staff)
  write (*,*) 'Staff (name,age,salary):', this % name, &
  this % age, this % salary
type is (employee)
  write (*,*) 'Employee (name,age,salary):', this % name, &
  this % age, this % salary
class is (person)
  write (*,*) 'Person (name,age):', this % name, this % age
class default
  stop 'Unknown person type'
end select
end subroutine write_person
```
Procedures as type components

In OO programming, procedures acting on the data are tied to the data.

Invocation of procedures for polymorphic variables via dynamic dispatch (runtime decision).

In Fortran 2003 procedures can be tied to
- objects (object-bound)
- types (type-bound)

The use of type-bound procedures is encouraged, as they enforce the object encapsulation and can be overridden.

Object-bound procedures

Object-bound procedures can be added as procedure pointer type components

```
procedure ([iface]) , pointer,[[, spec ;;] decl-list
```

where `iface` defines the procedure interface to be used and `spec` is defined as previously.

Unless initialized to be bound to an existing procedure, procedure pointer has to be initialized before it can be used.

Object-bound procedures example

```
type :: person
    character(len=10) :: name
    integer :: age
    procedure(print_per), pointer ::&
        print_person => null()
end type person

type, extends(person) :: employee
    integer :: salary
    procedure(print_emp), pointer ::&
        print_employee => null()
end type employee

subroutine print_per(this)
    class(person) :: this
    write (*,*) this % name, &
        this % age
end subroutine

subroutine print_emp(this)
    class(employee) :: this
    write (*,*) this % name, &
        this % age, &
        this % salary
end subroutine
```

Object-bound procedures example cont.

```
type(person) :: per
    type(employee) :: emp

! Initialize per and emp
per = person(name='Joe',age=10)
emp = employee(name='Mike',age=42,&
    salary=2000)

per % print_person => print_per
emp % print_person => print_per
emp % print_employee => print_emp

call per % print_person()
call emp % print_employee()
```
Type-bound procedures

Use of SELECT TYPE –
construct can be avoided
by overriding type-bound
procedures

Type-bound procedures
are dynamically deter-
mined for polymorphic
variables during runtime
and have a smaller
performance penalty than
a full type determination

Type-bound procedures

With type-bound procedures, type definition statement
includes a CONTAINS part
type [[,access][:: ] type_name
...contains
proc_def_stmt
end_type [type_name]
where proc_def_stmt defines the procedure or generic
tied to the type

Type-bound procedures cont.

Declaring a type-bound procedure with proc_def_stmt
procedure[(iface)][ [,attr_list][:: ] tbpname [=]pname
where each attribute attr in attr_list is one of
public or private
defered
non_overridable
nopass or pass[(arg_name)]

Procedure interface name iface defines the procedure
interface to be used and appears if and only if deferred
attribute appears

Procedure overriding

A way to add extend also the functionality of an
extended type is to override the procedures defined by
the base type

When extending a type, a procedure can be overriden by
simply defining a new procedure with the same name

Apart from the type of the passed-object dummy
argument which must be of the extended type,
overriding procedure must have exactly the same
interface as the overridden procedure

NON_OVERRIDABLE attribute to prevent overriding
Type-bound procedures example

```fortran
type :: person
  character(len=10) :: name
  integer :: age
contains
procedure, non_overridable :: &
  print_person => print_per
procedure :: &
  print_info => print_per
end type person

type, extends(person) :: employee
  integer :: salary
contains
procedure, non_overridable :: &
  print_employee => print_emp
procedure :: &
  print_info => print_emp
end type employee
```

```fortran
subroutine print_per(this)
  class(person) :: this
  write (*,*) this % name, &
           this % age
end subroutine

subroutine print_emp(this)
  class(employee) :: this
  write (*,*) this % name, &
           this % age, &
           this % salary
end subroutine
```

Type-bound procedures example cont.

```fortran
type(person) :: per

type(employee) :: emp

! Initialize per and emp
per = person(name='Joe', age=10)
emp = employee(name='Mike', age=42, &
               salary=2000)

! Calls print_per
call per  % print_info()

! Calls print_emp
call emp  % print_info()
```

Summary

- In the object-oriented programming paradigm data is coupled with the methods operating on the data.
- Fortran 2003/2008 have limited support the object-oriented programming model:
  - Type extensions
  - Type-bound procedures
  - SELECT TYPE -construct for determining the type during runtime
**Type access control**

In Fortran 95, outside the defining `module`, access to a type can be:

1. **Public**, where the type and all of its components are accessible.
2. For a type **public**, but **private** to the components of the type.
3. **Private**, where both the type and all of its components are hidden.

Using at least ii) is recommended for better encapsulation of the data.

**Type component access control in Fortran 2003**

Fortran 2003 allows mixed component accessibility for derived types. Type component definition statement:

```
def_stmt:
  type [[, attr_list] [,access]:] decl_list
```

where `access` is **public** or **private**.

Defining component accessibility overrides any default component access set to the type.

Accessibility for type-bound procedures in the `contains`—section of a derived type is declared independently of the component accessibility.
Module access control example

```fortran
module people_module
  type person
    private
    character(len=10), public :: name
    integer :: age
  contains
    procedure :: print_info => print_person ! Access to print_person via type is public
  private :: print_person ! Access to print_person via module is private
  contains
    ! Definition of print_person as before
  end type person
  private :: print_person
end module people_module
```

Module access control in Fortran 2003

- In addition to `public` and `private`, Fortran 2003 allows module variables to be defined `protected`
- A `protected` variable has
  - `public` visibility
  - Protection against modification outside the defining module similar to `intent(in)` attribute
- Can also be defined to be the default accessibility for all variables in a module similarly to `public` and `private`

Module access control example

```fortran
module access_module
  private ! Module default access is private
  integer, protected :: v1 ! v1 is protected
  integer :: v2 ! v2 is private
  integer, public :: v3 ! v3 is public
  public :: sub1
  contains
    subroutine sub1(...) ! sub1 is public
    end subroutine sub1
    subroutine sub2(...) ! sub2 is private
    end subroutine sub2
    subroutine sub3(...) ! sub3 is private
    end subroutine sub3
  end module access_module
```

Generic interfaces

```fortran
module mymodule
  type myreal
    real :: value
  end type
  interface operator(+)
    module procedure myadd
  end interface
  contains
    function myadd(x,y) result(z)
      implicit none
      type(myreal), intent(in) :: x, y
      type(myreal) :: z
      z % value = x % value + y % value
    end function myadd
    ! Other operations with myreal
  end module mymodule
```

- Modules can contain generic interfaces to procedures and operators
- Module operator overloading allows natural use of types
  - Automatic determination of routines to call already during compile time
Generic interfaces

- Fortran 95 allows defining and overloading generic names for procedures with module interface blocks

\[
\text{interface } [\text{generic\_spec}] \\
[\text{interface\_body}] \\
[\text{module\_procedure proc\_name\_list}] \\
\text{end\_interface } [\text{generic\_spec}]
\]

where \text{generic\_spec} is \text{generic\_name}, \text{operator}(\text{op}) or \text{assignment}(=)

- Visibility of the generic interface is always public

module procedure statement can only be present when \text{generic\_spec} is present

If \text{generic\_spec} is specified as \text{operator}, all the procedures within the block must be functions with one or two non-optional arguments with intent \text{in}

If \text{generic\_spec} is specified as \text{assignment}, all the procedures must be subroutines with two non-optional arguments, the first having intent \text{out} or \text{inout} and the second intent \text{in}

Procedure arguments must be uniquely determinable

Generic interface example

module access_module
   private  ! Module default access is private
   ...
   public :: sub1
   interface subs  ! subs interface is public
      module procedure sub2, sub3
   end interface subs
   contains
      subroutine sub1(...)  ! sub1 is public
      end subroutine sub1
      subroutine sub2(...)  ! sub2 is private
      end subroutine sub2
      subroutine sub3(...)  ! sub3 is private
      end subroutine sub3
   end module access_module

GENERIC type-bound procedures

In Fortran 2003 types can contain generic interfaces to procedures and operators

- When a type is accessible, all of its operators are accessible
  - Avoids programming errors from incorrect use of the \text{USE} statement

module mymodule
   type myreal
      real :: value
      contains
         procedure :: add => myadd
         generic :: operator(+) => add
   end type
   ! Other interfaces to operations contains
   function myadd(x,y) result(z)
      implicit none
      class(myreal), intent(in) :: x, y
      type(myreal) :: z
      z % value = x % value + y % value
      end function myadd
   ! Other operations with myreal
   end module mymodule
**GENERIC type-bound procedures**

- Type-bound procedures may be defined as generic with the **GENERIC** statement
  
  ```fortran
  generic [[,access] ::] generic_spec => tbp_name_list
  where generic_spec is as before and each tbp_name defines a type-bound procedure to be included in the generic set
  ```

- Procedures associated with the generic must not have the **nopass** attribute

- The extending type always extends the generic set

---

**GENERIC procedures example**

```fortran
module person_module
  type person
    private
    character(len=10) :: name
    integer :: age
  contains
    private
    procedure :: myname => get_name
    procedure :: myage => get_age
  generic, public :: &
    get => myname, myage
  end type person
  
  interface person
    module procedure create_person
  end interface
  contains
    function create_person(myname, &
      myage) result(per)
      character(len=10) :: myname
      integer :: myage
      type(person) :: per
      per = person(name=myname, age=myage)
    end function
    subroutine get_name(this, name)
      class(person), intent(in) :: this
      character(len=10) :: name
      name = this % name
    end subroutine get_name
    subroutine get_age(this, age)
      class(person), intent(in) :: this
      integer :: age
      age = this % age
    end subroutine get_age
  end interface
end module person_module
```

**GENERIC procedures example cont.**

```fortran
use person_module

type(person) :: t1
character(len=10) :: name
integer :: age

! Invokes create_person
! Invokes get_name
! Invokes get_age
```

---

**Unlimited polymorphic entities**

- References to any type (including intrinsic type) can be realized with unlimited polymorphic (UP) pointer

  ```fortran
  class(*), pointer :: name
  ```

- UP pointer can only be used as an actual argument, pointer target or **selector** in **select type** statement

- When an allocate statement is used with an UP pointer, required type and type parameters must be specified
Unlimited polymorphic entities example

```fortran
class(*), pointer :: v1, v2(:)
type(person), target :: p1
real, pointer :: r1(:)
v1 => p1
allocate(real :: v2(10))
select type (v2)
type is (real)
r1 => v2
class default
   stop 'Error in type selection'
end select
```

Abstract types

- An type can be **abstract** with **deferred** procedures having no actual implementation
  ```fortran
type, abstract :: abs_base
end type abs_base
```

- Extensions of an abstract type may be non-abstract
- A polymorphic variable can be declared to be of abstract type, but a variable cannot have an abstract type as its actual type
- Abstract types are commonly used in OO programming model to create common interfaces or to create programming hooks

Abstract types cont.

- Type can be **abstract** with no **deferred** procedures, but a type with **deferred** procedures must be **abstract**

```fortran
type, extends(abs_base) :: extended
integer :: field1
end type extended
```

```fortran
type(extended), target :: t1
class(abs_base), pointer :: p1
```

```
t1 = extended(i)
p1 => t1
```

Abstract types

- Abstract type creates a base type for all the extending types to build upon
- Abstract type can be used to create common interfaces for all the extending classes to implement
- Abstract types cannot be instantiated
### Abstract types example

```fortran
module person_module
  interface
    subroutine abs_print_if(this)
      import :: abstract_person
      class(abstract_person) :: this
    end subroutine abs_print_if
  end interface

  type, extends(abstract_person) :: person
  contains
    procedure :: print_person => print_per
  end type person

  subroutine print_per(this)
    class(person) :: this
    write (*,*) this % name, &
             this % age
  end subroutine
end module person_module
```

```fortran
use person_module
class(abstract_person), &
allocatable :: p1
p1 = person(name='Joe', age=10)
call p1 % print_person()
```

### FINAL components

- **Housekeeping (closing file handles, deallocating pointer components etc.)** may be needed when a type is deallocated
- **FINAL methods of a derived type** are automatically called whenever the type is deallocated.

**Subroutines of a derived type can be declared final**

```fortran
final [::] subroutine_name_list
```

where **subroutine_name** defines a set of **subroutines** one of which is run when the derived type is deallocated.

**subroutine_name** must be a subroutine with a single argument of the derived type, with uniquely defined parameter values and rank.

Final subroutines are not type-bound procedures and cannot be accessed through a type.
An object is **finalizable** if it has a final subroutine which matches the object.

A non-pointer object will be finalized when it is deallocated, goes out of scope, is passed to as an intent out argument or used as a left-hand side of an intrinsic assignment statement.

Derived type containing **finalizable** components will be finalized before the individual components.

Termination of a program does not invoke any **final** subroutines.

---

**Final components example**

```fortran
module final_module
  type :: final_type
  ...
contains
  final :: destroy_scalar_final, destroy_rank1_final
end type final_type
contains
  subroutine destroy_scalar_final(this)
    type(final_type) :: this
    ...
  end subroutine destroy_scalar_final
  subroutine destroy_rank1_final(this)
    type(final_type) :: this(:)
    ...
  end subroutine destroy_rank1_final
end module final_module
```

---

**Final components example cont.**

```fortran
subroutine sub1(n)
  integer :: n
  type(final_type) :: f1, f2(n)
  ...
  ! At the end of subroutine, destroy_scalar_final
  ! will be called for f1 and destroy_rank1_final for f2
end subroutine sub1
```

---

**FINAL components and extended types**

Extended derived types do not inherit any of the final subroutines of the old type.

Finalizing an extended derived type causes the parent type to be recursively finalized after the extended type has been finalized.
**Summary**

- In Fortran 2003/2008, mixed access control is allowed for better encapsulation of the methods of derived types.
- `GENERIC` interfaces can be added to modules and derived types to make use of mixed types easier and the code more readable.
- A polymorphic variable can be declared unlimited polymorphic to be type-compatible with any data.
- A derived type can be declared `abstract`.
- A derived type can have `FINAL` components.
A typical use case

- Problem of interoperability been present for long time
  - An application program maybe written entirely in Fortran
  - Suddenly a new library – written in C – becomes available
  - Will it be worthwhile to use such a new piece of code?
- How do we call it from Fortran?
- We will now show two ways of solving this problem:
  - Traditional, non-standard, de-facto way: “do not use this”
  - More modern, portable and recommended way

Traditional solution – “do not use this”

- Global Fortran subroutine and function names are normally viewed to the link-loader with routine name in lowercase letters followed by an underscore, e.g. `func_`
- INTEGER and REAL arguments are passed by-reference (i.e. by-address) to the memory region of the actual argument
- Fortran CHARACTER-strings are also passed by-reference, but argument list in C-language side must also contains so called hidden length of such array (int/long passed by-value)
- Note also that Fortran CHARACTER-strings are not null-character terminated as C-language would expect
Traditional solution (cont’d)

- Function return values usually work as expected, but data types must be matching between languages.
- Consider a test case, where we call a CBLAS-function to calculate single precision dot product.
- The known C-prototype from `<cblas.h>` is as follows:
  ```c
  float cblas_sdot (const int N,
                   const float *X, const int incX,
                   const float *Y, const int incY)
  ```
- In order to call this from Fortran – using traditional, not-recommended approach – we need an extra layer of “glue”.

Standard way (recommended)

- Whilst traditional way of binding Fortran to C did not look too bad, it is still potentially non-portable (a “hack”) and requires for example a-priori knowledge of lowercase and underscore business from the programmer.
- Furthermore, complex cases, like CHARACTER-string passing, or passing data by-value rather than by-reference may require quite a bit of programming “gymnastics”.
- With advent of the new standard it is possible to call aforementioned library function directly – “gluelessly”.
- The bottom line is to use INTERFACE-blocks and `bind(c)`
How to map data types and exchange data

- In the following slides we will cover briefly
  - Mapping of intrinsic data types by use of `ISO_C_BINDING`
  - Array variable interoperability
  - C-structures that can be mapped to Fortran derived types
How to access global data in terms of
- Fortran modules
- Fortran COMMON blocks
- Globally defined C-data

The ISO_C_BINDING module

- Activated – as we have already seen – via
  ```fortran
  USE, INTRINSIC :: ISO_C_BINDING
  ```
- Provides access to named constants that represent `kind` type parameters of data representations compatible with C types. Furthermore, the module also contains:
  - the derived type `C_PTR` corresponding to any C data pointer type
  - the derived type `C_FUNPTR` corresponding to any C func. pointer type
- Also contains few helper routines, e.g.
  - `C_LOC`, `C_FUNLOC`, `C_F_POINTER`, `C_ASSOCIATED`

Mapping between Fortran and C data types

- Using ISO C-bindings consistently, we can make sure our codes run correctly, despite “multi-lingual” nature
- The most common intrinsic data types mappings:

<table>
<thead>
<tr>
<th>Traditional “old” Fortran</th>
<th>Fortran declaration</th>
<th>C data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER*2</td>
<td>INTEGER(c_short)</td>
<td>short int</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>INTEGER(c_int)</td>
<td>int</td>
</tr>
<tr>
<td>INTEGER*8</td>
<td>INTEGER(c_long_long)</td>
<td>long long int</td>
</tr>
<tr>
<td>REAL*4</td>
<td>REAL(c_float)</td>
<td>float</td>
</tr>
<tr>
<td>REAL*8</td>
<td>REAL(c_double)</td>
<td>double</td>
</tr>
<tr>
<td>CHARACTER*1</td>
<td>CHARACTER(1,c_char)</td>
<td>char</td>
</tr>
</tbody>
</table>

Array data interoperability

- C-array indexing always starts from index zero (0)
- Fortran default indexing starts from one (1)
- Multidimensional arrays in C “grow fastest” along the last dimension e.g. 2D-arrays are called “row-major”
- Fortran multidimensional arrays are opposite, e.g. 2D-arrays are “column-major”
Passing character strings through ISO C bindings

There is a traditional “hack” how to pass character strings to C from Fortran, but since it is so susceptible to coding errors, we will not even show it – sorry!

Instead we show a handy example how to pass Fortran CHARACTER-string to a C standard library routine “atoi”
– Converts supplied characters into INTEGER representation
In this approach there are no worries about null-terminating strings.

Some additional remarks

A Fortran SUBROUTINE is mapped to a C-function with void result
– A Fortran FUNCTION on the other hand maps to a C-function that returns a value
The main program can be either in Fortran or C-language
– Link-loader must ensure though that Fortran RTL gets in
Binding <label> in bind(c, name=<label>)
– Is case sensitive when provided (e.g. name='C_funcX' )
– If the “name=" attribute is omitted, it takes Fortran name converted to lowercase (no underscores appended)

Accessing C data structures from Fortran

In many cases it is possible to describe Fortran derived data types in terms of C data structures (and vice versa)
You need ISO C binding module again plus Fortran derived type must have the bind(c) attribute
– However, use of sequence keyword is forbidden
Each individual Fortran data type component must also be of an interoperable type
Fortran components cannot be allocatables or pointers
C types cannot be unions nor structures with bit-fields
Accessing C data structures from Fortran

Note that variable ordering, data types and fixed array sizes must be identical. Typical usage comes through function calls, e.g. Fortran extracting information from a C-routine.

/* C data structure */
typedef struct {
  int count;
  double d[100];
} C_type;

/* C function example */
void c_func(C_type *p) {
  p->count = 1;
  p->d[0] = 1.23;
}

USE MY_TYPEDEF

SUBROUTINE TESTF(p) bind(c, name='C_func')
  TYPE(C_type_as_seen_by_Fortran) :: p
END SUBROUTINE TESTF

CALL TESTF(x)
PRINT *, x%count, x%d(1) ! NB: Fortran indices 1...100

Note: cannot map Fortran allocatable components

Please note that any attempt to use non-fixed size components and relate it to Fortran ALLOCATABLE is doomed because C and Fortran pointers are entirely different concepts.

Thus the following attempt does not work – period:

/* C data structure */
typedef struct {
  int count;
  double *d; // dynamic
} C_type;

Note: global data

Global data may be defined in Fortran in terms of data in the Fortran modules, or in COMMON – blocks.

In C-language they are declared once outside functions (often in main) and referenced via extern's elsewhere – These can be accessed in Fortran by use of bind(c) label.

Fortran module data is generally impossible to access from C as Fortran module names are compiler and linker dependent.

Consistently defined variables in Fortran COMMON-blocks can also be accessed in C-language side.
Some compatible global data mappings

```fortran
/* Global C data */
int number;
float Array[8];
double slice[3][7];
struct coord {
  float x, y, z;
};
struct coord xyz;
```

Handling (binary) I/O

- Restricting us to binary (unformatted) I/O only
  - Formatted text files are usually not a concern
- Fortran unformatted, non-direct access files by default contain record delimiters (4 or 8 bytes long)
  - Note: STREAM I/O and direct access files do not have them
- Files written from C-language don’t have record delims
- Could be a real headache
- Writing & reading files with STREAM I/O in Fortran usually solves most of the problems

The following data is hard for C-language to read

```fortran
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
INTEGER(kind = c_int) :: number
REAL(c_float) :: x, y, z
COMMON /xyz/ x, y, z
bind(c) :: /xyz/  ! Note /…/ syntax
END MODULE something
```

Using Fortran STREAM I/O solves our problems

```
MODULE something
USE , INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
INTEGER(c_int), bind(c) :: number
REAL(c_float) :: my_array(8)
bind(c, name='Array') :: my_array
REAL(c_double), bind(c) :: slice(7,3) | Index swap
REAL(c_float) :: x, y, z
COMMON /xyz/ x, y, z
bind(c) :: /xyz/  ! Note /…/ syntax
END MODULE something
```

Data written from Fortran as unformatted sequential file
- Contains record delimiters, usually 4-bytes at the beginning and end of each record
- Corresponding C-code has to “decipher” these delimiters

```
integer(kind = 4) :: ARRAY(100)
open (1, file='file.bin', form='unformatted', access='sequential', status='unknown')
write (1) ARRAY(1:100) ! Write 400 bytes + 8
write (1) ARRAY(1:50) ! Write 200 bytes + 8
close (1)
```

File ‘file.bin’ in a binary format, schematically:

```
400 | ... ARRAY(1:100) ... | 400
200 | ... ARRAY(1:50) ... | 200
```

File ‘file.bin’ now, schematically – it’s just data:

```
... ARRAY(1:100) ... ... ARRAY(1:50) ...
```

```
integer(kind = c_int) :: ARRAY(100)
open (1, FILE='file.bin', form='unformatted', access='STREAM', status='unknown')
write (1) ARRAY(1:100) ! Write 400 bytes
write (1) ARRAY(1:50) ! Write 200 bytes
close (1)
```

/* The corresponding C-reader is trivial */
```
int array[100];
FILE *fp = fopen("file.bin", "r");
```

```
fread (array, sizeof(*array), 100, fp);
fread (array, sizeof(*array), 50, fp);
fclose (fp)
```

The key: ACCESS = ‘STREAM’
No artificial record delimiters
Conclusions

- Fortran standard now officially supports mechanisms to call source codes or libraries written in C-language, as well as define Fortran routines to become callable from C
  - There seems to be a number of pitfalls
  - Use interoperability with care and avoid complicated structures and calling sequences

- Conflicting concept of pointers in Fortran and C makes matters even more difficult to comprehend

- When new STREAM I/O access is used in Fortran, binary files at least become interoperable with C-language
**BASIC CONCEPTS: PROCESS AND THREADS**

**Threads and processes**

**Process**
- Independent execution units
- Have their own state information and use their own address spaces

**Thread**
- A single process may contain multiple threads
- All threads within a process share the same state and same address space

**Process**
- Spawned when starting the parallel program and killed when its finished
- Typically communicate using MPI in supercomputers

**Thread**
- Short-lived: threads are created by forking and destroyed by joining them
- Communicate directly through the shared memory
WHAT IS OPENMP?

OpenMP

A collection of compiler directives and library routines that can be used for multi-threaded shared memory parallelization

- Fortran 77/9X and C/C++ are supported
- Current version implemented in most compilers is 3.0
- Most recent final version of the standard is 4.0 (released 23rd of July 2013)

Why would you want to learn OpenMP?

- OpenMP parallelized program can be run on your many-core workstation or on a node of a cluster
- Enables one to parallelize one part of the program at a time
  - Get some speedup with a limited investment in time
  - Efficient and well scaling code still requires effort
- Serial and OpenMP versions can easily coexist
- Hybrid programming

Three components of OpenMP

- Compiler directives and constructs
  - Expresses shared memory parallelization
  - Preceded by sentinel, can compile serial version
- Runtime library routines
  - Small number of library functions
  - Can be discarded in serial version via conditional compiling
- Environment variables
  - Specify the number of threads, etc.
OpenMP directives in Fortran

- Sentinels precede each OpenMP directive
- Fortran free form: !$OMP
- Fortran fixed form: c$OMP
  - Space in sixth column begins directive
  - No space depicts continuation line

OpenMP conditional compilation

- Conditional compilation with _OPENMP macro:
  #ifdef _OPENMP
  ! Thread specific code
  #else
  ! Serial code
  #endif
- Fortran fixed form guard sentinels: !$ *$ c$
- Fortran free form guard sentinels: !$

Compiling an OpenMP program

- Compilers that support OpenMP usually require an option that enables the feature
  - Cray: -homp
  - Intel: -openmp [-openmp-threadprivate=compat]
  - GNU: -fopenmp
  - PGI: -mp[=nonuma,align,allcores,bind]
  - Pathscale: -mp
- Without these options a serial version is compiled
  - Except on Cray, where compiling for OpenMP is the default

Example: Fortran90 Helloworld with OpenMP

```fortran
program hello
  use omp_lib
  integer :: omp_rank
  !$omp parallel private(omp_rank)
  omp_rank = 0
  !$omp_end parallel
end program hello
```

```
> ftn -mp omp_hello.f90 -o omp_hello
> setenv OMP_NUM_THREADS 4
> aprun -n 1 -d 4 ./omp_hello
Hello world! by thread 0
Hello world! by thread 2
Hello world! by thread 3
Hello world! by thread 1
```
PARALLEL REGIONS AND DATA SHARING

Parallel construct

- Defines a parallel region
  - Prior to it only one thread, master
  - Creates a team of threads: master+slave threads
  - At end of the block is a barrier and all shared data is synchronized

How do the threads interact?

- Because of the shared address space threads can communicate using shared variables
- Threads often need some private work space together with shared variables
  - For example the index variable of a loop
- Visibility of different variables is defined using data-sharing clauses in the parallel region definition

Default storage

- Most variables are shared by default
- Global variables are shared among threads
  - SAVE and MODULE variables, COMMON blocks
  - Dynamically allocated variables
- Private by default:
  - Stack variables of functions called from parallel region
  - Automatic variables within a block
Data-sharing attributes

**private(list)**
- Private variables are stored in the private stack of each thread
- Undefined initial value
- Undefined value after parallel region

**firstprivate(list)**
- Same as private variable, but with an initial value that is the same as the original objects defined outside the parallel region

**lastprivate(list)**
- Private variable
- The thread that performs the last parallel iteration step or section copies its value to the original object

**shared(list)**
- Comma separated list with shared variables
- All threads can write to, and read from a shared variable
- Variables are shared by default

**threadprivate(list)**
- Private global variable
- Commonly used to make global and module variables private to each thread
- Not allocated or initialized by default

**copyin(list)**
- Copy values of master thread’s threadprivate variables to threadprivate variables of other threads

**default(private/shared/none)**
- Sets default for variables to be shared, private or not defined
- default(none) can be useful for debugging as each variable has to be defined manually
Data sharing example

module datascope
integer :: A(5)
contains
subroutine work()
  integer :: B(2)
  !$omp parallel
  call dothings(B)
  !$omp end parallel
end subroutine work

subroutine dothings(var)
  integer :: var(2)
  double precision :: wrk(10)
  integer, save :: status
  ! ...
end subroutine dothings
end module datascope

Shared between threads
Private copy on each thread

WORK SHARING CONSTRUCTS

Work sharing

- Parallel region creates an "Single Program Multiple Data" instance where each thread executes the same code
- How can one split the work between the threads of a parallel region?
  - Loop construct
  - Single/Master construct
  - Sections
  - Task construct (in OpenMP 3.0 and above)
  - Workshare construct (with Fortran 90 array notation)

Loop constructs

- Directive instructing compiler to share the work of a loop
  - !$OMP DO
  - In order to be active, directive must be inside a parallel region
  - Can also be combined with parallel: !$OMP PARALLEL DO
- Loop index is private by default
- Work sharing can be controlled using schedule clause
**Loop scheduling clause**

- **schedule(static[,chunk])**
  - Blocks of iterations of size chunk for each thread

- **schedule(dynamic[,chunk])**
  - Chunk iterations off a queue until everything is done

- **schedule(guided[,chunk])**
  - Threads grab blocks of iterations, size of blocks starts from larger size and shrinks down to chunk

- **schedule(runtime)**
  - Schedule and chunk size are taken from environment variable OMP_SCHEDULE
Race condition

- Race conditions take place when multiple threads read and write a variable simultaneously, for example:

```fortran
asum = 0.0d0
!$OMP PARALLEL DO SHARED(x,y,n,asum) PRIVATE(i)
do i = 1, n
  asum = asum + x(i)*y(i)
end do
!$OMP END PARALLEL DO
```

- Random results depending on the order the threads access `asum`
- We need some mechanism to control the access

Reductions

- Summing elements of an array is an example of reduction operation:

\[ S = \sum_{j=1}^{N} A_j = \sum_{j=1}^{\lfloor \frac{N}{2} \rfloor} A_j + \sum_{j=\lfloor \frac{N}{2} \rfloor+1}^{N} A_j = B_1 + B_2 = \sum_{j=1}^{i} B_j \]

- OpenMP provides support for common reductions with the reduction clause
Reduction clause

`reduction(operator:var_list)`

- Performs reduction on the (scalar) variables in list
- Private reduction variable is created for each thread’s partial result
- Private reduction variable is initialized to operator’s initial value
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable

Reduction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>.AND.</td>
<td>.true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.false.</td>
</tr>
<tr>
<td>.NEGV.</td>
<td>.false.</td>
</tr>
<tr>
<td>.IEOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IAND.</td>
<td>All bits on</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.true.</td>
</tr>
<tr>
<td>MIN</td>
<td>max pos.</td>
</tr>
<tr>
<td>MAX</td>
<td>min neg.</td>
</tr>
</tbody>
</table>

Race condition example revisited

```c
 !$OMP PARALLEL DO SHARED(x,y,n) PRIVATE(i) REDUCTION(+:asum)
 DO i = 1, n
     asum = asum + x(i)*y(i)
 END DO
 !$OMP END PARALLEL DO
```

EXECUTION CONTROLS AND SYNCHRONIZATION
Execution controls

Sometimes a part of parallel region should be executed only by the master thread or by a single thread at time
  – IO, initializations, updating global values, etc.
  – Remember the synchronization!

OpenMP provides clauses for controlling the execution of code blocks

**master**
  – Specifies a region that should be executed only by the master thread
  – No implicit barrier at the end of construct

**single**
  – Specifies that a region should be executed only by a single (arbitrary) thread
  – Implicit barrier at the end of construct

**barrier**
  – Synchronizes all threads at this point
  – When a thread reaches a barrier it only continues after all threads have reached it
  – Implicit barrier at: end of do, parallel, sections, single, workshare
  – Restrictions:
    ▪ Each barrier must be encountered by all threads in a team, or none at all
    ▪ The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team

**critical** [(name)]
  – A section that is executed by only one thread at a time
  – Optional name specifies global identifier for critical section
  – Unnamed critical sections are treated as the same section

**flush** [(name)]
  – Synchronizes the memory of all threads
  – Makes sure each thread has a consistent view of memory
  – Implicit flush at
    ▪ All explicit and implicit barriers
    ▪ Entry to / exit from critical section and lock routines
Execution controls

**atomic**
- Strictly limited construct to update a single value, cannot be applied to code blocks
- Only guarantees atomic update, does not protect function calls
- Can be faster on hardware platforms that support atomic updates

Example: reduction using critical section

```plaintext
!$OMP PARALLEL SHARED(x,y,n,asum) PRIVATE(i, psum)
psum = 0.0d
!$OMP DO
  do i = 1, n
    psum = psum + x(i)*y(i)
  end do
!$OMP END DO
!$OMP CRITICAL(dosum)
  asum = asum + psum
!$OMP END CRITICAL(dosum)
!$OMP END PARALLEL
```

Example: initialization and output

```plaintext
!$OMP PARALLEL
  do while (err > tolerance)
    !$OMP MASTER
      err = 0D0
    !$OMP END MASTER
    !$OMP BARRIER
    ! Compute error
    !$OMP SINGLE
      write (*,'(A,ES7.2)') 'Error is now: ', err
    !$OMP END SINGLE
  end do
!$OMP END PARALLEL
```

Example: updating global variable

```plaintext
integer :: a(100), globalmax, localmax, i
! ...
globalmax = 0
localmax = 0
! ...
!$OMP PARALLEL FIRSTPRIVATE(localmax) PRIVATE(i)
!$OMP DO
  do i=1,100
    localmax = MAX(localmax, a(i))
  end do
!$OMP END DO
!$OMP CRITICAL(domax)
globalmax = MAX(localmax, globalmax)
!$OMP END CRITICAL(domax)
!$OMP END PARALLEL
```
OpenMP and execution environment

- OpenMP provides several means to interact with the execution environment. These operations include:
  - Setting the number of threads for parallel regions
  - Requesting the number of CPUs
  - Changing the default scheduling for work-sharing clauses
  - etc.

- Improves portability of OpenMP programs between different architectures (number of CPUs, etc.)

Environment variables

- OpenMP standard defines a set of environment variables that all implementations have to support
- The environment variables are set before the program execution and they are read during program start-up
  - Changing them during the execution has no effect
- We have already used OMP_NUM_THREADS

Runtime functions

- Runtime functions can be used either to read the settings or to set (override) the values
- Function definitions are inside
  - `omp_lib` Fortran module (`omp_lib.h` header in some implementations)
- Two useful routines for distributing work load:
  - `omp_get_num_threads()`
  - `omp_get_thread_num()`
List of runtime routines (OpenMP 3.1)

Fortran:

LOGICAL FUNCTION OMP_IN_FINAL()
SUBROUTINE OMP_SET_NUM_THREADS(num_threads)
   INTEGER num_threads
SUBROUTINE OMP_GET_NUM_THREADS()
SUBROUTINE OMP_GET_MAX_THREADS()
SUBROUTINE OMP_GET_THREAD_NUM()
SUBROUTINE OMP_GET_NUM_PROCS()
LOGICAL FUNCTION OMP_IN_PARALLEL()
SUBROUTINE OMP_SET_DYNAMIC(dynamic_threads)
   LOGICAL dynamic_threads
LOGICAL FUNCTION OMP_GET_DYNAMIC()
SUBROUTINE OMP_SET_NESTED(nested)
   LOGICAL nested
LOGICAL FUNCTION OMP_GET_NESTED()
SUBROUTINE OMP_SET_SCHEDULE(kind,modifier)
   INTEGER(KIND=omp_sched_t) kind
   INTEGER modifier
SUBROUTINE OMP_GET_SCHEDULE(kind,modifier)
   INTEGER(KIND=omp_sched_t) kind
   INTEGER modifier
SUBROUTINE OMP_GET_THREAD_LIMIT()
SUBROUTINE OMP_SET_MAX_ACTIVE_LEVELS(max_active_levels)
   INTEGER max_active_levels
SUBROUTINE OMP_GET_MAX_ACTIVE_LEVELS()
INTEGER FUNCTION OMP_GET_ACTIVE_LEVEL()
INTEGER FUNCTION OMP_GET_LEVEL()
INTEGER FUNCTION OMP_GET_ANCESTOR_THREAD_NUM(num)
   INTEGER num
SUBROUTINE OMP_INIT_LOCK(lock)
   INTEGER (KIND=OMP_LOCK_KIND)::lock
SUBROUTINE OMP_DESTROY_LOCK(lock)
   INTEGER(KIND=OMP_LOCK_KIND)::lock
SUBROUTINE OMP_SET_LOCK(lock)
   INTEGER(KIND=OMP_LOCK_KIND)::lock
SUBROUTINE OMP_UNSET_LOCK(lock)
   INTEGER(KIND=OMP_LOCK_KIND)::lock
LOGICAL OMP_TEST_LOCK(lock)
   INTEGER(KIND=OMP_LOCK_KIND)::lock
SUBROUTINE OMP_INIT_NEST_LOCK(lock)
   INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock
SUBROUTINE OMP_DESTROY_NEST_LOCK(lock)
   INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock
SUBROUTINE OMP_SET_NEST_LOCK(lock)
   INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock
SUBROUTINE OMP_UNSET_NEST_LOCK(lock)
   INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock
INTEGER OMP_TEST_NEST_LOCK(lock)
   INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
DOUBLE PRECISION FUNCTION OMP_GET_WTICK()

List of environment variables (OpenMP 3.1)

OMP_SCHEDULE
OMP_NUM_THREADS
OMP_DYNAMIC
OMP_PROC_BIND
OMP_NESTED
OMP_STACKSIZE
OMP_WAIT_POLICY
OMP_MAX_ACTIVE_LEVELS
OMP_THREAD_LIMIT

Parallelizing loop with library functions

!$OMP PARALLEL PRIVATE(i, nthr, tid, nL)
   nthr = omp_get_num_threads()
   tid = omp_get_thread_num()
   nL = n/nthr
   do i=(nL*tid)+1,(nL*(tid+1))
      ! ...
   end do
!$OMP END PARALLEL

FURTHER TOPICS
OpenMP programming best practices

- Maximize parallel regions
  - Reduce fork-join overhead, e.g., combine multiple parallel loops into one large parallel region
  - Potential for better cache re-usage
- Parallelize outermost loops if possible
  - Move `DO` construct outside of inner loops
- Reduce access to shared data
  - Possibly make small arrays private

Things that we did not cover

- Other work-sharing clauses
  - Sections
  - Tasks
  - Workshare
- More advanced ways to reduce synchronization overhead with `nowait` and `flush`
- Nested parallelism
- Hardware related topics (affinity, false sharing, etc.)

OpenMP summary

- OpenMP is an API for thread-based parallelization
  - Compiler directives, runtime API, environment variables
  - Relatively easy to get started but specially efficient and/or real-world parallelization non-trivial
- Features touched in this intro
  - Parallel regions, data-sharing attributes
  - Work-sharing and scheduling directives
Web resources

- OpenMP homepage
  http://openmp.org/

- Good online tutorial:
  https://computing.llnl.gov/tutorials/openMP/

- More online tutorials:
  http://openmp.org/wp/resources/#Tutorials
Work sharing: workshare

- Parallel execution of program regions written with Fortran 90 array syntax: !$OMP WORKSHARE
- Restrictions: code block can only contain
  - array and scalar assignments
  - FORALL statements and constructs
  - WHERE statements and constructs
  - OpenMP atomic, critical or parallel constructs
- No user function calls, except ELEMENTAL

Work sharing: workshare example

```fortran
real A(100,100), B(100,100)
call random_number(A)
!$OMP PARALLEL SHARED(A, B)
!$OMP WORKSHARE
where(A < 0.5)
    B = 0.0
elsewhere
    B = 1.0
end where
!$OMP END WORKSHARE
!$OMP END PARALLEL
```
Work sharing: sections

- Parallel execution of independent code regions, each executed by one thread: !$OMP SECTIONS, !$OMP SECTION
- Commonly used to assign different subroutine calls to different threads
- Difficult to load balance
  - Number of threads v.s. number of sections

Work sharing: sections example

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
call do_a()
!$OMP SECTION
call do_b()
!$OMP SECTION
call do_c()
!$OMP END SECTIONS
!$OMP END PARALLEL
```

Work sharing: task

- Task construct assigns tasks to threads: !$OMP TASK
- A task has
  - Code to execute
  - Data environment
  - A thread executing it
- Task queue is managed by OpenMP runtime
Work sharing: task

Task data scoping follows some of the rules for parallel regions
- static and global variables are shared
- local variables are private

Main exception: variables defined in the enclosing context are by default firstprivate unless set shared

---

Task data scoping example

```fortran
![Example code]
```

---

Work sharing: task

Tasks created by any thread of the current thread team can be expected to be complete at
- OpenMP barrier (implicit or explicit) exit

Task barrier construct taskwait: !$OMP TASKWAIT

Task encountering taskwait suspends until its child tasks are complete

taskwait only applies direct children generated by the enclosing task, not to descendants of children

---

Work sharing: task

Task is tied to the thread that first executes it. For tied tasks, the scheduling constraints are
- A task can only be suspended at a suspend point (Task creation, Task finish, barrier, taskwait)
- If a task is not suspended, executing thread can only switch to a direct descendant of a task tied to the thread

Task can be declared untied
- No scheduling restrictions, can be suspended at any point
- Tasks can migrate between threads, breaking thread centric constructs (i.e., threadprivate, critical, etc.)
Work sharing: task example

```fortran
integer :: x
!
$OMP PARALLEL
$OMP SINGLE
x = fib(n)
$OMP END SINGLE
$OMP END PARALLEL
```

Recursive function `fib(n)` result(`fn`)

```fortran
integer :: j, n, fnm, fn
if (n<2) then
  fn = n
  return
else if (n<10) then
  fn = fib_seq(n)
  return
end if
$OMP TASK SHARED(fn)
fn = fib(n-1)
$OMP END TASK
$OMP TASK shared(fnm)
fnm = fib(n-2)
$OMP END TASK
$OMP TASKWAIT
fn = fn + fnm
end function fib
```

Execution control: locks

- Two types of locks available: simple and nested
  - Simple lock may only be locked if in an unlocked state
    `INTEGER(kind=omp_lock_kind) svar`
  - Nested lock may be locked multiple times by the same thread
    `INTEGER(kind=omp_nest_lock_kind) svar`
- Locks enable
  - Implementation of asynchronous behaviour
  - Unstructured execution control
- Must be manipulated through the OpenMP API only
- If not initialised, the behaviour is undefined

Locks provide functionality similar to semaphores

- General syntax for basic functionality:
  ```fortran
  SUBROUTINE OMP_FUNC_LOCK(svar)
  func expresses functionality: init, destroy, set,
  unset, init_nest, destroy_nest, set_nest,
  unset_nest
  END SUBROUTINE
  ```
- Functions to test and set lock if lockable:
  ```fortran
  LOGICAL OMP_TEST_LOCK(svar)
  INTEGER OMP_TEST_NEST_LOCK(svar)
  ```
- It is often preferrable to use `critical` or `atomic`
Execution control: locks

Workflow:
1. Define a lock variable
2. Initialise with `omp_init_lock`
3. Set with `omp_set_lock` or `omp_test_lock`
4. Release with `omp_unset_lock`
5. Destroy with `omp_destroy_lock`

Execution control: locks example

```fortran
integer(kind=omp_lock_kind) lock
call omp_init_lock(lock)
!$OMP PARALLEL
  do while (.not. omp_test_lock(lock))
    ! Do something else
  end do
  ! Critical work here
  call omp_unset_lock(lock)
!$OMP END PARALLEL
call omp_destroy_lock(lock)
```

MEMORY HIERARCHY, AFFINITY

Many current systems consist of nodes that have more than one CPU socket per node:
- Reduced cost due to shared peripherals and connectors
- Fast communication between sockets
- Larger shared memory for hybrid programs
NUMA architectures

- Non Uniform Memory Access
  - All memory is accessible, but latencies and bandwidths vary
- ccNUMA = NUMA with cache coherency
- For example the nodes in Taito: CPUs with their own memory controllers and fast interconnects (Intel QPI)

NUMA and OpenMP

- OpenMP 3.X does not have NUMA support
  - The standard does not determine where the data is stored
  - NUMA control with utilities such as `numactl` or `taskset`
- Common allocation policy is "first touch": the thread initializing data will host the data in its local memory
  - Initialization loops where threads write initial data using same access pattern as computation parts
- A single MPI process per socket often preferred

Thread affinity

- The operating system assigns threads and processes to physical cores
  - In Linux the default is "soft affinity": OS tries to avoid moving threads from one core to another
- For the best computational performance it is useful to bind (pin) threads to specific cores
  - Even more important in NUMA architectures: remember the data allocations

OpenMP and thread affinity

- The affinity settings are implementation and environment specific, for example
  - GNU: `GOMP_CPU_AFFINITY="0-6"`
  - Intel: `KMP_AFFINITY=proclist=[0-6],explicit`
  - PGI: `MP_BIND=y; MP_BLIST=0,1,2,3,4,5,6`
- Batch job system and its settings
  - Some systems have affinity control also at this level
**Cache Coherency**

- Modern CPUs have complicated caches
- Data is read and written as full cache lines (often 64 bytes)
- Programming models require that the data in memory has to be consistent (one address can only have one value)
  - Cache coherency logic is needed to ensure the consistency in multicore architectures

**False Sharing**

- When different threads modify values in successive memory locations, cache coherency forces these updates to be transferred between all cache copies
  - If this happens in a rapid succession there is a huge performance penalty due to cache misses
- How to avoid performance penalty?
  - Does not happen when data is only read
  - Reorganize the data access so that each thread modifies values inside a larger block or use private variables

**False Sharing Example**

```fortran
integer :: i, TID, n, nthr, count, counter(nthr)
double precision :: vec(n)
count = 0
counter = 0
!
$OMP PARALLEL PRIVATE(i,TID)
  TID=1
  !$OMP DO
  do i=1,n
    if (vec(i)<0) counter(TID)=counter(TID)+1
  end do
  !$OMP END DO
$OMP END PARALLEL
!
Threads | Time (s)
---|---
1 | 1.13
2 | 0.76
4 | 1.06
8 | 0.59
```

Different threads doing lots of writes to successive addresses
**False sharing example, corrected**

```fortran
integer :: i, TID, n, threads, count, Lcount
double precision :: vec(n)
count = 0
counter = 0

!$OMP PARALLEL PRIVATE(i, Lcount, TID)
TID=1
!$ TID=omp_get_thread_num()+1
Lcount = 0
!$OMP DO
do i=1,n
   if (vec(i)<0) Lcount=Lcount+1
end do
!$OMP END DO
!$OMP ATOMIC
count = Lcount+count
!$OMP END PARALLEL

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.05</td>
</tr>
<tr>
<td>2</td>
<td>0.53</td>
</tr>
<tr>
<td>4</td>
<td>0.26</td>
</tr>
<tr>
<td>8</td>
<td>0.13</td>
</tr>
</tbody>
</table>
```

No conflicting writes between threads.
What is Co-Array Fortran (CAF) about?

- Adds parallel processing as part of Fortran language
  - Only small changes required to convert existing Fortran code to support a robust and potentially efficient parallelism
- A Partitioned Global Address Space (PGAS) language
  - Unlike OpenMP, which operates over shared memory, the CAF extensions implement parallelism over “distributed shared memory” → CAF is potentially massively parallel
- Has been integrated into Fortran 2008 standard
  - Caveat: only few compilers so far support the CAF syntax – never mind producing efficient object code

Some history

- Co-Array Fortran (formerly known as F-- i.e. Fortran-minus-minus as opposed to C++) was created by Bob Numrich and John Reid in the 1990s
- Reached the current form in 1998 as a simple extension to Fortran95 for parallel processing
- CAF has many years experience mainly on Cray hardware
- A set of CAF’s core features are now part of the Fortran 2008 standard (ISO/IEC 1539-1:2010)

CAF does ...

- Add only a few new rules to the Fortran language
- Provide mechanisms to allow
  - SPMD (Single Program, Multiple Data) style of explicitly parallel programming
  - Data distribution over partitioned memory (you can think about “distributed shared memory” here)
  - Guard against race conditions (in e.g. variable value assignments) by using synchronization
  - Memory management for dynamic shared entities
Execution model

- Upon startup a Co-Array Fortran program gets replicated into a number of copies called images (i.e. processes)
  - The number of images is usually decided at the execution time, but (in rare cases) can also be fixed at compile time
- Each “replica” (image) runs asynchronously in a loosely coupled way until program controlled synchronization
- Image’s (local) data are visible within the image only – except for data declared as special arrays i.e. co-arrays
- One-sided data communication enables movement of co-array data across different images of a CAF program

Data distribution through co-arrays

- CAF programs become meaningful in parallel programming context, when their data are remotely accessible by its images
- Accomplished through additional Fortran syntax for co-arrays for Fortran arrays or scalars, for example:

  ```fortran
  INTEGER, CODIMENSION[*] :: scalar
  INTEGER :: scalar[*]
  REAL, DIMENSION(64), CODIMENSION[*] :: vector
  REAL :: vector(64)[*]
  ```

- Declares a scalar with one “incarnation” on every image
- Declares a vector with 64 elements on every image

Data distribution through co-arrays ...

- The square brackets [*] denote allocation of special arrays over allocated images (decided upon program startup)
  - Think of the asterisk “*” being replaced by the num_images()
  - The round brackets “( )” mean local array accesses, and the “[ ]” are meant for remote data array accesses only

```
! Your very first Co-Array Fortran program!

PROGRAM HELLO_WORLD
IMPLICIT NONE
WRITE(*,'("Hello world from image ",i0," out of ",i0)') &
  this_image(), num_images()
END PROGRAM HELLO_WORLD
```

- num_images() returns the number of images in use for this run (usually set outside the program, by the environment)
- this_image() returns the image number in concern – numbered from one (not zero) to num_images()
- This program is a trivially parallel i.e. each image does not explicitly share any data and runs seemingly independently
A further peek into vector(64)[*]

Each image has a local vector with 64 elements
- A program has total \( \text{NPES} = \text{num}\_\text{images}() \) images

For image "me = this\_image()", local access to the element \( j \) either by “vector\%(j\)" or “vector\%(j\)[me]"

Synchronization – avoiding race conditions

- We need to be careful when updating co-arrays
  - Is the remote data we are copying valid e.g. up to date?
  - Could another image overwrite our data without notice?
  - How do we know if data sent to us has already arrived?

CAF provides synchronization statements, e.g. adds barrier for synchronization of all images:

\[
\text{SYNC ALL}
\]

To be absolutely sure we are getting correct result, we need to modify our previous copying example a little ...

Synchronization : corrected remote copy

- We need to add barrier synchronization of all images before the copying takes place to be absolutely sure we are getting the most up to date copy of \( \text{GLOBAL}():[\text{NPES}] \)

In this particular case – since only the image \# NPES – is involved, we could use an alternative form of synchronization:

\[
\text{GLOBAL}(): = \text{this}\_\text{image()} * (/ 1, 2, 3 /) \text{ Local initialization}
\]

\[
\text{SYNC IMAGES ( NPES )}
\]

\[
\text{LOCAL}(): = \text{GLOBAL}():[\text{NPES}] \text{ A copy ("broadcast") from image number "NPES" to each image}
\]

Another synchronization example : data input

- CAF assumes by default that standard input is connected to the first image only – other images open /dev/null for input

In order to replicate input data (e.g. a scalar input integer), the following code snippet becomes handy – but remember to synchronize before starting to use your (now local) data

\[
\text{REAL(kind = 8)} :: \text{PARAM}[*] \text{ A scalar – one value per image}
\]

\[
\text{IF ( this\_image() = 1 ) THEN}
\]

\[
\text{READ *,PARAM} \text{ Read just one value i.e. PARAM[1]}
\]

\[
\text{DO } j\text{roc} = 2, \text{num_images}()
\]

\[
\text{PARAM[jroc] = PARAM } \text{ The same as PARAM[jroc] = PARAM[1], but “faster”}
\]

\[
\text{ENDDO}
\]

\[
\text{ENDIF}
\]

\[
\text{SOME\_VALUE = PARAM * 3.14}
\]

Don’t forget the following barrier synchronization or your data maybe out of sync

\[
\text{SYNC ALL}
\]

\[
\text{SOME\_VALUE = PARAM * 3.14}
\]
Viewing global data with co-arrays

In many computational problems it is handy to think of having large global array(s), where important data reside.

Since Fortran does not provide global view, a programmer needs to be creative: CAF offers a fantastic help here.

In the next slides we show how to map into a global view ("global array") in a linear fashion.

You must have noticed already at this point, that a co-array in each image ought to have the same (local) size!

- In another lecture we circumvent this by use of Fortran TYPEs.

Global view: linear mapping across images

Create an example co-array with 3 elements per image.

We pretend it being global in the sense that the first 3 elements are on image#1, the next 3 on image#2 and so on.

We want to access global data in a linear fashion, like

\[ GA(global\_index) := \text{CAF\_GA(local\_index)}[\text{owner\_image}] \]

To accomplish this kind of virtual global array of \( GA \), we need to compute \((\text{local\_index},\text{owner\_image})\)–pair from each given \( global\_index \) – and uniquely.

The following table illustrates mapping of global index:

<table>
<thead>
<tr>
<th>Owner_image</th>
<th>Local_index</th>
<th>Global_index</th>
<th>Data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>
**Summary**

- In this lecture we covered basic aspects of co-arrays:
  - How to declare co-dimensional arrays and access data
  - Concept of images and some related functions
  - Importance of synchronization
  - How to implement simple linear global view of data

We also touched how to use standard input and output:
- So it is now possible to write simple CAF programs!!

In the subsequent lecture we will also get acquainted ourselves to multidimensional and allocatable co-arrays

---

```
INTEGER, PARAMETER :: NEL = 3 ! Number of elements on each image
REAL(KIND=8) :: CAF_GA(NEL) ! Our “global” array, but defined as a co-array
INTEGER :: NPES ! Number of images
INTEGER :: NGA ! Global length of GA i.e. NEL x NPES
INTEGER :: GLOBAL_INDEX ! Global index, between 1 and NGA
INTEGER :: LOCAL_INDEX ! Local indexing, on each image
INTEGER :: OWNER_IMAGE ! Image that “owns” global index

NPES = NUM_IMAGES() ! Initialize CAF_GA somehow on each image first and then remember to synchronize
NGA = NEL * NPES
CAF_GA(1:NEL) = (/ (LOCAL_INDEX, LOCAL_INDEX=1,NEL) /) * THIS_IMAGE() ! Access some global data as if we were accessing global array “GA(global_index)”
SYNC ALL

GLOBAL_INDEX = 7
IF (GLOBAL_INDEX >= 1 .and. GLOBAL_INDEX <= NGA) THEN
  LOCAL_INDEX = MOD(GLOBAL_INDEX - 1,NEL) + 1
  OWNER_IMAGE = (GLOBAL_INDEX + NPES - 1)/NPES
  SOME_GA = CAF_GA(LOCAL_INDEX)(OWNER_IMAGE)
ENDIF
```

---

Global view: linear mapping across images...

In this lecture we covered basic aspects of co-arrays:
- How to declare co-dimensional arrays and access data
- Concept of images and some related functions
- Importance of synchronization
- How to implement simple linear global view of data

We also touched how to use standard input and output
- So it is now possible to write simple CAF programs!!

In the subsequent lecture we will also get acquainted ourselves to multidimensional and allocatable co-arrays
More Co-Array Fortran features

- Operating with multiple co-dimensions
- Co-array subscripts, functions `lcobound()` & `ucobound()`
- Use of `image_index()` – function, and more on `this_image()`
- Dynamic allocation across co-dimensions
- Variable length co-arrays
- Co-arrays in procedures

Recap: 1-dimensional co-array

- We refer to variables with co-dimension (or co-rank) being 1
- The declaration statement was either of the following:
  ```fortran
  INTEGER :: A(N)*
  INTEGER, codimension[*] :: A(N)
  ```
- The size of array (the “N”) had to be the same on each image
- The co-index [ ] ends up being trivially the same as image index

<table>
<thead>
<tr>
<th>Image#1</th>
<th>Image#2</th>
<th>Image#3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2) [1]</td>
<td>A(2) [2]</td>
<td>A(2) [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multiple co-dimensions

- Now co-rank > 1, thus there is more than one co-dimension
  - Things are getting (much) more complicated to figure out
  - The last co-dim. must always be declared with asterisk “*”
- Sum of all dimensions (rank plus co-rank) must be ≤ 15
- For simplicity, let’s declare a co-array scalar with two co-dimensions i.e. co-rank is 2, and first co-dimension as 4
- The syntax is either of the following:
  ```fortran
  REAL :: S [4, *]
  REAL, codimension[4,*] :: S
  ```
REAL :: S[4,*]

Suppose we have allocated 10 images for the CAF-program.
Numbers in the green slots denote images holding data.
References to the red ones produce invalid co-indices (= 0).

Co-array subscripts

Co-array declaration by default assumes normal Fortran array declaration rules.
- Lower co-bound for each dimension starts from 1.
- Upper co-bound denotes the size of corresponding co-dimension.

But as with conventional Fortran arrays we can declare the lower co-bound be different from 1.

The upper co-bound – as long as it is not the last co-dimension – can also be different from size.
- Upper bound for the last co-dimension must always be "*".

Array & co-array declarations with subscripts ...

Fortran INTEGER array, with 2 ranks, lower bounds at (1,1) and upper bounds at (7,3), and shape of (7,3), is declared in usual way:

INTEGER, dimension(7,3) :: IA

A REAL-valued co-array scalar, with co-rank 1, lower co-bound of zero, and upper co-bound no more than num_images():

REAL, codimension[0:*] :: Scalar

A REAL-valued co-array, with rank of 4 and co-rank 3; lower bounds at (-2,0,1,-1) and lower co-bounds at (-3,2,10); upper bounds at (5,5,7,4) and upper co-bounds at (6,4,10-1+n); shape of (8,6,7,6) and co-shape of (10,3,n), with n = ceiling(num_images())/(10*3).

REAL :: Array(-2:5,0:5,7,−1:4) [−3:6, 2:4, 10:*]

REAL :: S[4,*]

Instead of a scalar valued co-array we could have a multi-dimensional (regular) Fortran array.
- Each box would then represent image’s local array.

The syntax is again either of the following, f.ex.:

REAL :: A(3,3)[4,*] :: A(3,3)

Multiple co-dimensions ...

REAL :: S[3,2] – a single scalar value of S.


S[4,*]
Obtaining lower and upper bounds for co-arrays

- Often complicated to figure out upper/lower co-bounds
- Luckily there are query functions for obtaining them:
  - `lcobound(coarray [, dim])`
  - `ucobound(coarray [, dim])`
- They return an integer array, or in case of a single dimensional query (with optional “dim”) – a scalar

For the `Array` from the previous page:
- `lcobound(Array)` produces output `-3  2  10`
- `ucobound(Array, dim=2)` gives `4`

Diving into the last co-dimension

- Consider a co-array with co-rank of 4
  - `INTEGER :: Scalar [p,q,*,r]`
- Lower co-bounds would all be equal to 1
- All but the last co-dimension has upper co-bounds of `(p,q,r)`
- The last co-dimension acts like an assumed shape array i.e. the asterisk “*” can be replaced by a ceiling value of `num_images() / (p * q * r)`
- This is defined at runtime and always becomes ≥ 1

Use of `image_index()` and `this_image()`

- So far we have seen `this_image()` function been used without arguments
  - Returns the image (“process”) number where the CAF-program is being run
- In its another calling form it takes a co-array as an argument, e.g.
  - `REAL :: A(7,7) [0:3,3:*]`
  - `print *, this_image(A)`
- Running with 10 images again it returns (say) for image#7 a `vector [ 2, 4 ]` i.e. co-indices of CAF [ ] –brackets expression

  ![Image Index Diagram]

  ```plaintext
  0 1 5 9
  2 6 10
  3 7 0
  4 8 10

  A(:,;)[2,4]
  ```

- `image_index()` performs the reverse of `this_image()`
- Takes co-indices (e.g. [2, 4] on A-array from previous slide)
  - Returns the owner image number in concern (e.g. image#7)
  - Useful in synchronization, e.g. `SYNC IMAGE (image_list)`
- In particular, returns zero (0), when the given co-indices are out of bounds i.e. they do not belong to any image

  ```plaintext
  INTEGER :: img
  img = image_index(A, (/ 2, 4 /)) ! Will return 7
  img = image_index(A, [3, 5]) ! Returns 0 – no owner image
  ```
ALLOCATABLE co-arrays

- It is possible to define fully dynamic co-arrays, where both shape (i.e. locally owned part) and co-shape are dynamic, e.g. an allocatable with deferred shapes of rank 1 and co-rank 3:
  ```fortran
  INTEGER, allocatable :: A(:) [:, :, :]
  ALLOCATE (A(1000)[4,20,-5:*])
  DEALLOCATE (A)
  ```
- ALLOCATE and DEALLOCATE imply implicit synchronization – all images must participate i.e. an implicit SYNC ALL occurs
- The local size (here: 1000) must be the same on each image
- The last co-dimension must have an asterisk “*”

About POINTERs with co-arrays

- A co-array declaration cannot have a POINTER attribute
  Thus the following would be illegal co-array declaration:
  ```fortran
  REAL, POINTER :: ptr[:,:]
  ```
  This is invalid Fortran
- However, we can define a new data TYPE, where type component(s) have POINTER (or ALLOCATABLE) attributes
  - And then define a co-array being of that TYPE
- Used in dynamic dimensioning of co-arrays
  - This way local sizes on every image can be different

Variable length co-arrays via ALLOCATABLE

- Create a new Fortran data type with ALLOCATABLE component in it – and place it in a MODULE file:
  ```fortran
  TYPE mytype
    REAL, ALLOCATABLE :: data(:)
  END TYPE mytype
  ```
- Then declare a co-array of that type, e.g. `co`-rank of 1
  ```fortran
  TYPE (mytype) :: co[*]
  ```
- Make `co % data` to point different chunks of local data
  ```fortran
  REAL, TARGET :: chunk(1000)
  CALL get_my_range(size(chunk), istart, iend)
  co % data => chunk(istart:iend)
  ```

Variable length co-arrays via POINTER

- Create a new Fortran data type with a Pointer component
  ```fortran
  TYPE mytype
    REAL, POINTER :: data(:)
  END TYPE mytype
  ```
- Then declare a co-array of that type, e.g. co-rank of 1
  ```fortran
  TYPE (mytype) :: co[*]
  ```
  A fully legal construct
- Make `co % data` to point different chunks of local data
  ```fortran
  REAL, TARGET :: chunk(1000)
  CALL get_my_range(size(chunk), istart, iend)
  co % data => chunk(istart:iend)
Co-arrays in procedures

- When declared in a subroutine or function, a co-array must be one of the following:
  - Declared as a dummy argument to the procedure
  - Have ALLOCATABLE and/or SAVE attribute
- Re-mapping of co-rank is also allowed
- A co-array in procedure cannot be an automatic array—like by nature: the “limitation” stems from the fact local size of a co-array on each image must be the same.

Assumed size Co-arrays

- Co-arrays can also be declared as assumed size i.e. co-dimension(s) can be re-mapped to have a different co-rank in the procedure than was declared in the caller, e.g.

```fortran
SUBROUTINE SOME_ROUTINE (N, ARRAY_1, CO_ARRAY_1, ARRAY_2, CO_ARRAY_2, CO_ARRAY_3)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: M, N
INTEGER, INTENT(INOUT) :: ARRAY_1(N)         ! Explicit shape
INTEGER, INTENT(INOUT) :: CO_ARRAY_1(N)      ! Explicit shape
INTEGER, INTENT(INOUT) :: CO_ARRAY_2(1000)   ! Assumed shape
INTEGER, INTENT(INOUT) :: CO_ARRAY_3(1000)   ! Assumed shape
INTEGER, INTENT(INOUT) :: LOCAL_CO_ARRAY_1(N)         ! Ok, co-array with SAVE
INTEGER, INTENT(INOUT) :: LOCAL_CO_ARRAY_2(1000)   ! Ok, co-array with SAVE-attr
INTEGER, INTENT(INOUT) :: LOCAL_CO_ARRAY_3(1000)   ! Ok, co-array with SAVE-attr
INTEGER :: LOCAL_ARRAY_1(1000)                   ! Ok, automatic (regular) array
INTEGER :: LOCAL_ARRAY_2(1000)                   ! Ok, local (regular) array
INTEGER :: LOCAL_CO_ARRAY_1(1000)               ! Invalid: co-array can’t be automatic
INTEGER :: LOCAL_CO_ARRAY_2(1000)               ! Invalid: SAVE-attribute missing
INTEGER :: LOCAL_CO_ARRAY_3(1000)               ! Ok, co-array with SAVE-attr
IMPLICIT NONE
END SUBROUTINE SOME_ROUTINE
```

Summary

- In this lecture we have dived deeper into the CAF world
- Use of multiple co-dimensions can be hard to understand
  - Using CAF specific intrinsic functions `lcobound`, `ucobound`, `this_image` and `image_index` usually clarify the situation
- Co-arrays can also be dynamic through ALLOCATABLE
  - Direct use of POINTER attribute is, however, forbidden
- Variable size co-arrays can be defined via TYPE definition
  - trick: use of POINTER as TYPE component is then allowed
- Co-arrays can also appear in procedures, but specific rules must be adhered, e.g. adding SAVE/ALLOCATABLE attributes
Advanced CAF features

- Is the array-syntax over co-dimensions supported?
- More advanced synchronization clauses
- Miscellaneous CAF features

Array-syntax across co-dimensions?

One clearly strange thing in CAF-syntax is that use of array syntax over co-dimensions is not allowed
- It used to be possible prior to CAF becoming part of Fortran2008 std !

Suppose we want to gather data over all images into a local array, one element per image. It would be tempting to do:

```fortran
REAL :: global_data[*]
REAL :: local(num_images()) ! Even this is illegal
CALL initialize_somehow(global_data)
SYNC ALL
local(:) = global_data[:]
```

Array-syntax [:] across co-dimensions is not allowed

Instead – an explicit looping over images is mandatory

```fortran
REAL :: global_data[*]
INTEGER :: jroc
REAL, allocatable :: local(:)
ALLOCATE (local(num_images()))
CALL initialize_somehow(global_data)
SYNC ALL
! With some further optimization
DO jroc = 1, this_image() - 1
   local(jroc) = global_data[jroc]
ENDDO
local(this_image()) = global_data
DO jroc = this_image() + 1, num_images()
   local(jroc) = global_data[jroc]
ENDDO
```
More advanced synchronization

- CRITICAL sections
- SYNC IMAGES
- SYNC MEMORY
- Atomic updates with `atomic_define` and `atomic_ref`
- LOCK variables

At times it is necessary to execute a block of code by one image at a time to avoid race conditions whilst updating co-arrays at a fixed image location, e.g. [P]

```fortran
PROGRAM CRIT
IMPLICIT NONE
INTEGER :: MYSUM[*] = 0
INTEGER :: ANS ! The reference result
INTEGER :: ME, NPES, P
P = 1 ! Fixed image# [P]
ME = THIS_IMAGE()
NPES = NUM_IMAGES()
ANS = (NPES * (NPES + 1))/2
! SYNC ALL : usually needed, but not here
CRITICAL
  MYSUM[P] = MYSUM[P] + ME
END CRITICAL
SYNC ALL  !! Must have !!
PRINT *,ME,': MYSUM[P], ANS=',MYSUM[P],ANS
END PROGRAM CRIT
```

SYNC IMAGES

There are situations when only a subset of images need to synchronize; use of `SYNC ALL` would be an overkill

- `SYNC IMAGES (image_list)` provides a desired interface
  - Not all images need to call this
- The `image_list` can be one or more image ids, or "*
  - The list does not have to be same on every image
- Special `image_list` is asterisk "*" and denotes all images
  - But the `SYNC IMAGES (*)` is not `SYNC ALL`
  - Image pair-wise synchronization is faster than `SYNC ALL`

SYNC IMAGES – examples

The 1st image synchronizes with all the other images in a pair-wise manner

- Only the two first images (an image set) synchronize – other images are not involved at all in (this) synchronization
SYNC MEMORY

- Enforces any pending co-array writes to finish before proceeding
- Waits until any preceding writes are visible to the images in concern
- Flushes also any cached reads on image's co-array(s)
- Not actual synchronization at all, however SYNC ALL / SYNC IMAGES also imply SYNC MEMORY

ATOMIC-updates

- The standard includes two functions for ATOMIC updates in co-arrays:
  - `ATOMIC_DEFINE`
  - `ATOMIC_REF`

  Use with `SYNC MEMORY` to implement spin-wait between images P and Q
  - A “producer-consumer”
  - Take a notice of ISO Fortran environment — module

LOCK variables

- In fact it is recommended to avoid both SYNC MEMORY and ATOMIC–update routines and use LOCK – UNLOCK –constructs
- The aforementioned spin-wait program now looks much cleaner
  - The LOCK_TYPE is taken again from ISO Fortran env.

LOCK variables

- Use of LOCK variables resembles entering to a CRITICAL section, except that it is more flexible, since you can have many distinct LOCKs active simultaneously — unlike just one with a CRITICAL section
- You can also acquire a LOCK in a non-blocking manner by requesting “acquire_lock” from LOCK construct
**Miscellaneous CAF features**

- **I/O conventions in CAF**
- **Program termination**
  - Including STAT= and ERRMSG= specifiers in synchronization
- **Future CAF**
  - Technical specification (TS)

**I/O conventions in CAF**

- The default output (“stdout”) is connected to all images
  - Output is merged (in any order) into one output stream
  - PRINT *, WRITE(*,...), WRITE(6,...)
- The standard error (“stderr”) is redirected to the “stdout”
  - WRITE(0,...)

**Program termination**

- Normal termination of a CAF program is initiated by STOP and END PROGRAM statements on all images
- In problematic cases, however, one or more images “in trouble” can request for termination, whilst other images continue to run until reaching some synchronization point
  - Image data of the images being in an abnormal termination state are still available and can be requested by other images
- The remaining images could test the termination status of the other image(s) by using STAT= specifier in synchronization statements, and also get ERRMSG—string
- ERROR STOP statement on one or all images will initiate an error termination and all images are expected to be killed shortly
A run, where the 1st image is in trouble in ALLOCATE

Test image status against STAT_STOPPED_IMAGE (from ISO Fortran env.)

- Available in SYNC ALL,
  SYNC IMAGES, LOCK,
  UNLOCK & SYNC MEMORY

The remaining image can now proceed with fetching CAF data – still available

**Technical specification (TS)**

- Co-array Fortran continues to evolve
- Technical specification outlines the future developments
- A useful document to read

In the pipeline are the following extensions to the CAF

- Image teams
- Failed images
- Collective intrinsic functions for co-arrays
- Improvements in atomic updates

**TS: Image teams**

- Learning lessons from MPI–standard and its communicators, it is useful to have subsets of images forming “teams” of processes
- Define a new (“sub”-)team at runtime
  - All images with the same “id” e.g. `mod(this_image(), 2)` join
  - `FORM SUBTEAM (id, Ateam)`
- Synchronize within a team only
  - `SYNC TEAM (Ateam)`

**TS: Failed images**

- An image may have died during the run – consequently a `SYNC IMAGES` would fail (or stall) if it contains the failed image
  - `SYNC IMAGES (image_list, stat = st)`
  - If `(st == stat_failed_image)` then
- `failed_images()` function returns the count of failed images at the given moment
  - Another possibility is to use `num_images(failed = .TRUE.)`
- Used with care – and together with the new team capability, a fail-safe computing becomes possible (still non-trivial)
TS: Collectives

- Collective operations are essential part of //-programs
  - Broadcast data
  - Calculate global sums, maximums, etc.
- Available in MPI and OpenMP standards
  - MPI_Allreduce and OpenMP reduction operations
- Routines for example
  - CO_BROADCAST
  - CO_SUM, CO_MIN, CO_MAX
  - CO_REDUCE

TS: Atomic operations

- CRITICAL or LOCK statements often an overkill for
  \( \text{count}[1] = \text{count}[1] + 1 \)
- For example OpenMP enables atomic operations
  ```
  !$OMP atomic
  shared_count = shared_count + 1
  ```
- Proposed coding:
  ```
  INTEGER (atomic_int_kind) :: count[*]
  CALL atomic_define (count[1], 0)
  CALL atomic_add (count[1], 1)
  CALL atomic_ref (thevalue, count[1])
  ```

Summary

- We have now completed our excursion to CAF world
- In the last “chapter” we were told that the common array syntax is not extended to co-dimensions – which is odd
- We got better insight to more advanced synchronization across images
- We also had a recap on Fortran I/O and standard channels
- We learned how to terminate images abnormally and yet gain access to almost destroyed co-array data : fail safety
- Finally, we had a browse on to future of CAF and its TS