A brief overview of this course

- Advanced Fortran Programming concentrates on aspects related to the most recent versions of the Fortran standard (F2003/2008/2015)
  - some Fortran 95 less-known features are covered, too
- Our major F2003/F2008 subjects in this course are
  - Language interoperability
  - Object-oriented Fortran (OOF)
  - Parallel programming with Fortran coarrays

Evolution of Fortran standards over the few decades

- The 1st version became a standard by John Backus (IBM/1957)
- Mid-60’s the most significant version was labeled as ANSI standard, called FORTRAN66
- 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

Evolution of Fortran standards...

- An important extension to FORTRAN77 was release of “military standard” Fortran enhancements (also in 1978) by US DoD, and adopted by most FORTRAN77 compilers
  - IMPLICIT NONE
  - DO – END DO
  - INCLUDE statement
  - Bit manipulation functions
- All these were eventually incorporated to the next major official standard release – Fortran90
A major step in keeping Fortran programming alive was introduction of the Fortran 90 standard in ’91 (ISO), ’92 (ANSI)

- Free format source input (up to 132 characters per line)
- Dynamic memory handling

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

A minor revision in 1997 (ISO) some features were 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-INTEGRERS as DO loop parameters
- H edit descriptor in FORMATS
- ASSIGN and assigned GOTO

Fortran 2003 was a significant revision of the Fortran 95 standard (introduced in 2004)
- Object-oriented programming (OOP), e.g.
  - Type extension, (single) inheritance, polymorphism, ...
- Parameterized derived types
- Procedure POINTERs
- Interoperability with C (and other) languages
- OS interfacing: command line arguments & env. variables
- ALLOCATABLE improvements
- POINTER improvements
- I0 edit descriptor for INTEGRERS in FORMAT statements

Fortran 2008: A minor revision of Fortran 2003 (approved 2010)

New features include
- Coarrays
- Submodules
- CONTIGUOUS attribute
- BLOCK construct
- newunit= in OPEN statement
- DO CONCURRENT construct
- G0 edit descriptor for REALs in FORMAT statements
Useful new features since Fortran 90

Outline

- I0 and G0 descriptors for dynamic output formatting
- Interaction with the operating system
  - Accessing command line arguments and environment variables
  - Running operating system commands from Fortran
- CONTIGUOUS attribute
- Asynchronous I/O
- Enhancements in use of ALLOCATABLE variables

I0 and G0 edit descriptors

- Dynamic sizing of real and integer valued output
  - Correspond to C language %d and %g formats
- Output fields become left justified with all the unnecessary leading blanks (and precision for reals) removed

```fortran
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 <i=12345, reals=1.230000 1.234567890000000 >
```
INTERACTION WITH THE OPERATING SYSTEM

Command line arguments

- Parameters to a program are very often given to programs as command line arguments
  - Input file(s), modify program behavior, etc.
- Besides command line arguments, environment variables are a common way to influence program behavior
- Fortran 2003 introduced a standardized method for
  - reading command line arguments
  - accessing values of environment variables

Access separate command line arguments
get_command_argument(number[,value][,length][,status])
  - number denotes which argument to get
  - value is a character string that contains the value of the requested argument on return (optional)
  - length contains the length of the requested argument on return (optional)

Get the number of command line arguments
integer :: command_argument_count()
Command line input

Example: reading in two integer values from the command line, e.g.

```
% ./a.out 100 100
```

```fortran
subroutine read_command_line(height, width)
    integer, intent(out) :: height, width
    character(len=10) :: args(2)
    integer :: n_args, i
    n_args = command_argument_count()
    if ( n_args /= 2 ) then
        write(*,*) 'Usage : ./exe height width'
        call abort()
    end if
    do i = 1, 2
        call get_command_argument(i, args(i))
        args(i) = trim(adjustl(args(i)))
    end do
    read(args(1),*) height
    read(args(2),*) width
end subroutine read_command_line
```

Environment variables

Access a value of an environment variable

```
call get_environment_variable(name, value[, length] [, status][, trim_name])
```

- `name` is a character string that contains the name of the requested variable
- `value` is a character string that contains the value of the requested variable
- `length` is the length of the requested variable on return (optional)
- `trim_name` is of type logical and sets if trailing blanks are allowed in variable names or not (optional)

```
program environment
    implicit none
    character(len=256) :: enval
    integer:: len,stat
    ! extract hostname
    call get_environment_variable('hostname', enval, len, stat)
    if (stat == 0) write (*,'(a,a)') 'host=', enval(1:len)
    ! extract user
    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 sometimes useful

- No source nor library API available for a useful program
- perl/python/etc parsing scripts

Fortran 2008 has a standardized method for invoking an external command
Executing commands

- Execute a command line
  
  ```fortran
  call execute_command_line(command[,wait]
      [,exitstat][,cmdstat][,cmdmsg])
  ```

  - `command` is a character string containing the command
  - `wait` is logical value indicating if command termination is to be waited (optional)
  - `exitstat` is an integer value containing the return value of the command if `wait=.true.` (optional)
  - `cmdstat` is zero if `command` executed successfully (optional)
  - `cmdmsg` is a character string containing explanatory message for positive values of `cmdstat` (optional)

Executing commands: example

```fortran
program execcommand
  implicit none
  integer :: estat, cstat
  ! execute a unix command
  call execute_command_line('ls -al', .true., estat, cstat)
  if (estat==0) write (*,'(a)') 'command completed successfully'
end program execcommand
```

CONTIGUOUS attribute

- Unit stride data access (contiguous) is significantly faster than a constant non-unit stride (non-contiguous)
- Typical optimization: use structure of arrays instead 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
  ```
CONTIGUOUS attribute

- In Fortran 2008, an assumed shape or a pointer variable can be declared \textit{contiguous}
  
  \begin{verbatim}
  real, pointer, contiguous :: x(:)
  real, contiguous :: y(:,:)
  \end{verbatim}

- Testing contiguity
  
  \begin{verbatim}
  logical :: is_contiguous(arr)
  \end{verbatim}
  
  \textit{arr} is an array of any type
  
  - the function returns \texttt{.true.} if \textit{arr} is contiguous and \texttt{.false.} otherwise

- Potential performance benefits
  
  - Array traversal and element address computation is simplified
  - Improved vectorization properties
  - No need to generate auto-dispatch code for contiguous data separately
  - Simply contiguous: no need to declare the array as contiguous
    - A whole array of that is not of assumed shape or pointer
    - Continuous section of a contiguous array

- program contigtest
  
  \begin{verbatim}
  implicit none
  integer, parameter :: m = 6, n = 5
  real, target :: A(m,n)
  real, pointer :: A_flat(:), A_diag(:)
  real, allocatable, contiguous :: b(:)

  A = 0                        ! A is simply contiguous
  A_flat(1:m*n) => A           ! A_flat is simply contiguous
  A_diag => A_flat(:,m+1)      ! A_diag is non-contiguous
  A_diag = 1                   ! b is contiguous

  allocate(b(m))               ! b is contiguous
  write (*,'(L,L,L,L)') is_contiguous(A), is_contiguous(A_flat), &
                  is_contiguous(A_diag), is_contiguous(b)

  ! Outputs 'T,T,F,T'
  end program contigtest
  \end{verbatim}
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

If you open your file for asynchronous I/O, but perform actual I/O operations in other procedure, then also the arrays involved in the I/O must also be declared with the asynchronous attribute

Use the **asynchronous** attribute in open, read and/or write
- When all I/O statements are performed in the same routine, then the variables need not to be declared asynchronous

Variables cannot be accessed whilst being processed by async I/O

ENHANCEMENTS TO ALLOCATE
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 do not have save attribute as well.

The Fortran 90 standard introduced user-defined datatypes, but dynamic allocation was restricted to pointers only. The restriction was removed in Fortran 2003. Allocation remains intact even after going out of scope, if the type was defined in a module.

A function can have the allocatable attribute. Works similarly as when allocatable is a dummy argument.

An allocatable variable can now appear in the procedure argument list. Enables dynamic memory allocation in the called routine based on sizing information. Allocated (and perhaps initialized) data is returned to the caller. The caller must remember to deallocate.

The caller contains:

- allocate(ka(nk))
- call get_distinct_values(x,...,ka,nk)
- deallocate(ka)

A function can have the allocatable attribute. Works similarly as when allocatable is a dummy argument

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 do not have save attribute as well.
Assigning values to an allocatable variable

Assigning values to an allocatable array 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
May require additional compiler flags, like –assume realloc_lhs (Intel) and –ew (Cray)

```
program test
    real, allocatable :: a(:,), b(:)
    real :: input(3)
    read *, input ! read 3 values from stdin
    a = input ! automatic allocate
    ! a(:, ) = input(:, ) would be illegal!
    print *, size(a) ! gives 3
    b = (/ 1, 2, 4, 8, 16 /) ! automatic alloc.
    a = [ a, b ] ! automatic extent ok
    print *, size(a) ! size now = 8
    ! deallocate legal, but not necessary
deallocate(a, b)
end program test
```

Allocatable scalars

These are especially useful with Fortran character strings, whose length is decided at run time
Allocation can either explicit or automatic (see previous slide)
May require additional compiler flags due to implicit assignments of allocatables

```
program test
    character(:,), allocatable :: ch, s
    integer :: ch_len
    ! explicit allocation
    ch_len = length('hello world')
    allocate(character(ch_len) :: ch)
    ch(:, ) = 'hello world'
    ! implicit allocation
    ! (s(:, ) = ... fails)
    s = 'hello world'
    print *, length(ch), length(s) ! prints 12 12
    ! deallocate legal, but not required
deallocate(ch, s)
end program test
```

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

```
program test
    integer, allocatable :: a(:,), b(:)
    allocate (a(1:5)); a(:, ) = 0
    a(3) = 3
    print *, 'a=',a ! output: 0 0 3 0 0
    call move_alloc(a,b)
    ! a is now deallocated & b allocated
    if (allocated(a)) print *, 'a=',a
    if (allocated(b)) print *, 'b=',b
    ! no need for explicit deallocates
deallocate(a)
! would be illegal: deallocate(a)
end program test
```

Two improvements to the pointer assignments

It is possible to set the desired lower bounds of a pointer to any value, e.g.:
real, target :: state_budget(1917:2016)
real, pointer, dimension(:) :: a, b, c
a => state_budget
b => state_budget (1939:1945)
c(1939:) => state_budget (1939:1945) ! new
Two improvements to the pointer assignments

The target of a multi-dimensional array pointer can be a one-dimensional array

```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)
```

After this the pointer array `matrix` operates across all the memory region that has been allocated, and the pointer `diagonal` refers to the diagonal elements

Summary

- It is now possible to obtain command line arguments, access environment variables and run operating system commands in a standardized way
- Contiguous attribute for arrays enables more optimization
- Asynchronous I/O enables overlap with computation
- New edit descriptors enable dynamic sizing of output
- Dynamic memory allocation has also been improved by means of enhancements to allocatable variables and pointers
Language interoperability issues

The problem of language interoperability has been present for a long time

- Before Fortran2003 the standard did not specify the calling convention in detail

Interoperability is becoming even more important than before

- Steering of computation with scripting languages
- Utilizing libraries that are written using other programming languages

Before Fortran2003

Compilers often, but not always

- Passed all procedure arguments by-reference (i.e. by-address)
- Referred to functions and subroutines in lowercase letters and by adding an additional underscore after, e.g. `Func` becomes `func_` to be found by the linker
- Passed function return values via stack
- Passed `CHARACTER`-strings by-reference, with an additional hidden length argument, passed by value. NOTE: Fortran `CHARACTER`-strings are not null-character terminated
Portability

The traditional way to have interoperability with C requires a-priori knowledge of lowercase and underscore policy used by the compiler.

Complex cases, such as passing CHARACTER–strings or passing arguments by value, are generally very error prone and may lead to catastrophic errors at runtime.

Often a separate layer of C functions was needed for interoperability.

The ISO_C_BINDING module

Fortran 2003 intrinsic module ISO_C_BINDING is used—as we have already seen—with

USE, INTRINSIC :: ISO_C_BINDING

Module contains

– Access to named constants that represent kind type parameters of data representations compatible with C-types.
– The derived types C_PTR and C_FUNPTR corresponding to C pointer and C function pointer types, respectively
– Useful procedures: C_LOC, C_FUNLOC, C_F_POINTER, C_ASSOCIATED, C_F_FUNPOINTER, C_SIZEOF (F08)

Interoperability with Fortran 2003: example

```
program F_calls_C
  use, intrinsic :: iso_c_binding
  implicit none
  integer(kind=C_INT), PARAMETER :: n = 8
  real(kind=C_DOUBLE), dimension(n) :: plm
  integer :: err
  interface
    function gsl_plm_array(lmax, m, x, res) &
      & bind(c, name='gsl_sf_legendre_Plm_array') result(rval)
      integer(kind=C_INT) :: rval
      integer(kind=C_INT), value :: lmax, m ! Pass by value
      real(kind=C_DOUBLE), value, intent(in) :: x ! intent(in)=const float
      real(kind=C_DOUBLE) :: res(*)
    end function gsl_plm_array
  end interface
  err = gsl_plm_array(n+1, 2, 0.234_C_DOUBLE, plm)
  print *, plm
end program F_calls_C
```

Calling C routines

A Fortran SUBROUTINE maps to a C-function with void result.

A Fortran FUNCTION maps to a C-function returning a value.

Binding label in bind(c, name=<label>)

– The routine is known to the C compiler as specified by the binding label.
– By default the Fortran name in lower case. If provided, case sensitive ignoring leading and trailing blanks (name='C_funcX')
Mapping of C intrinsic data types

Interoperable mappings for the most commonly used C intrinsic data types

<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)</td>
<td>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>

Note that Fortran does not support unsigned integers

VALUE attribute

For scalar dummy arguments, VALUE attribute can be used to pass the value to the procedure

– No copy is made from the procedure to the caller when the call returns
– Argument must not be a POINTER, ALLOCATABLE, have intent(IN) or intent(INOUT), be a procedure or have the VOLATILE attribute

NOTE: VALUE attribute is not limited to procedures with the bind attribute

Mapping of derived data types

In many cases it is possible to describe Fortran derived data types in terms of C data structures (and vice versa)

To be interoperable, Fortran derived type must have the bind(c) attribute

– sequence or extends keywords are forbidden

Individual Fortran components in the data type must be of an interoperable type

– zero-sized array components are forbidden

– ALLOCATABLE and POINTER components are forbidden

C types cannot be unions nor structures with bit-fields

Mapping of derived data types

For a derived type to be interoperable between C and Fortran, variable ordering, data types and array sizes must be identical

– Variable names do not need to be identical

Typical usage comes through function calls, for instance a Fortran routine extracting information from a C routine
Mapping of derived data types: example

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

type(C_type_in_Fortran) :: X
call testf(X)
write (*,*) X % count, X % d(1)
end program typedeftest

Array indexing
– C: starts from zero (0)
– Fortran: by default, starts from one (1)
Multidimensional ordering
– C: grows fastest along the last dimension
– Fortran: “column-major”, grows fastest along the first dimension

Mapping of array data

/* Corresponding C-declarations */
double z1[5], z2[17][3];
int ivec[12];

Mapping of character data

C procedures expect character strings to be null terminated
– A null character has to be appended to a string before calling the C procedure

Module ISO_C_BINDING contains many character constants: C_ALERT, C_BACKSPACE, C_FORM_FEED, C_CARRIAGE_RETURN, C_HORIZONTAL_TAB, C_NULL_CHAR, C_NEW_LINE, C_VERTICAL_TAB

Mapping of character data: example

program F_atoi
use, intrinsic :: ISO_C_BINDING
implicit none
interface
! ... c_null_char
number = trim(year) // c_null_char
write (*,'(A,I0)') 'atoi('//year//')=',atoi(number)
end program F_atoi

Array indexing
– C: starts from zero (0)
– Fortran: by default, starts from one (1)

Multidimensional ordering
– C: “row-major”, grow fastest along the last dimension
– Fortran: “column-major”, grow fastest along the first dimension

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

type(C_type_in_Fortran) :: X
call testf(X)
write (*,*) X % count, X % d(1)
end program typedeftest

program F_atoi
use, intrinsic :: ISO_C_BINDING
implicit none
interface
! ... c_null_char
number = trim(year) // c_null_char
write (*,'(A,I0)') 'atoi('//year//')=',atoi(number)
end program F_atoi

program F_atoi
use, intrinsic :: ISO_C_BINDING
implicit none
interface
! ... c_null_char
number = trim(year) // c_null_char
write (*,'(A,I0)') 'atoi('//year//')=',atoi(number)
end program F_atoi

Array indexing
– C: starts from zero (0)
– Fortran: by default, starts from one (1)
Multidimensional ordering
– C: “row-major”, grow fastest along the last dimension
– Fortran: “column-major”, grow fastest along the first dimension

Mapping of derived data types: example

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

type(C_type_in_Fortran) :: X
call testf(X)
write (*,*) X % count, X % d(1)
end program typedeftest

Array indexing
– C: starts from zero (0)
– Fortran: by default, starts from one (1)
Multidimensional ordering
– C: “row-major”, grow fastest along the last dimension
– Fortran: “column-major”, grow fastest along the first dimension

Mapping of derived data types: example

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

type(C_type_in_Fortran) :: X
call testf(X)
write (*,*) X % count, X % d(1)
end program typedeftest

Picture of an empty page
Accessing global data

Global data:
- Fortran: is declared in modules or in COMMON blocks
- C: is declared outside any local scope and referenced via extern elsewhere

C global data is accessed in Fortran with bind(c, name=<label>)
- Optional name-parameter is defined similarly as when accessing C procedures
- NOTE: using bind(c) implies save

Interoperability with C pointers

Pointer concept is completely different in Fortran and C
- Fortran: POINTER is a storage descriptor which holds an address of the TARGET object and, for arrays, the related bounds and strides
- C: pointer is an address to a memory location

Module ISO_C_BINDING contains Fortran type definitions for C pointers and C function pointers
- TYPE(C_PTR) abstracts C pointers
- TYPE(C_FUNPTR) abstracts C function pointers

Interoperability with C pointers

ISO_C_BINDING contains inquiry and manipulation functions for TYPE(C_PTR) and TYPE(C_FUNPTR)

When calling C routines with pointer arguments, C storage locations are sometimes needed. These can be acquired with c_loc and c_funloc:

- TYPE(C_PTR) FUNCTION c_loc(x)
- TYPE(C_FUNPTR) FUNCTION c_funloc(x)

where x in interoperable.

- For an interoperable Fortran entity x, return TYPE(C_PTR) (or TYPE(C_FUNPTR)) containing the C address of x
Interoperability with C pointers

Checking association status of C pointer can be done with `c_associated` function:

```fortran
LOGICAL FUNCTION c_associated(ptr1[, ptr2])
where ptr1 and ptr2 are TYPE(C_PTR) or TYPE(C_FUNPTR)

- Return .FALSE. if ptr1 is a NULL C pointer or if ptr2 is present and has a different value. Otherwise return .TRUE.
```

Interoperability with C pointers

When access to the actual storage elements is needed, the contents of C pointer must be first transferred to Fortran form. This can be done with `c_f_pointer` routine:

```fortran
SUBROUTINE c_f_pointer(c_p, f_p [, shape])
where c_p is scalar TYPE(C_PTR), f_p is a Fortran POINTER. If present shape is a rank-1 integer array with size equal to rank of f_p

- Set f_p to point to c_p. If f_p is an array, shape must be present. When f_p is an array, its lower bounds are set to 1 and upper bounds according to shape.
```

Interoperability with C pointers: example

```fortran
/* C data structure */
typedef struct {
int count;           
double *d; /* dynamic */
} SimpleType;

/* C-function example */
#include <stdlib.h>   // for malloc
void my_func(SimpleType *p, int n) {
  p->count = n;
  p->d = malloc(n * sizeof(*p->d));
  if (n > 0) p->d[0] = 1.23;
}

MODULE my_typedef
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
TYPE, bind(c) :: simpletype
integer(kind=c_int) :: count
type(c_ptr) :: d
END TYPE simpletype
INTERFACE
SUBROUTINE MY_FUNC(P,N) bind(c,name='my_func')
  IMPORT TYPE(simpletype)
  INTEGER(c_int), value :: N
  CALL MY_FUNC(X, N)
END SUBROUTINE MY_FUNC
END INTERFACE
END MODULE my_typedef
```

```fortran
PROGRAM mapcpointer
USE my_typedef
TYPE(simpletype) :: X
INTEGER(c_int), PARAMETER :: N = 10
REAL(c_double), POINTER :: xd(:)
CALL MY_FUNC(X, N)
CALL C_F_pointer(x % d, xd, (/ N /)) ! Map data to Fortran pointer
PRINT *,X % count, xd(1)
END PROGRAM mapcpointer
```
Interoperating with C binary I/O

Interoperability problems extend to binary (unformatted) I/O
- Fortran: unformatted I/O files contain data and the associated record delimiters (4 or 8 bytes long)
- C: binary I/O files contain data

Writing and reading files with STREAM I/O in Fortran solves data interoperability problems
- Stream I/O files do not have the facility to backspace or skip over records

Interoperating with C binary I/O: example

```
INTEGER(kind = c_int) :: ARRAY(1:100)
OPEN(11, FILE='file.bin', &
form='unformatted', &
access='sequential', &
status='unknown')

! Write 400 bytes + 8
WRITE (11) ARRAY(1:100)
! Write 200 bytes + 8
WRITE (11) ARRAY(1:50)
CLOSE (11)
```

Data written from Fortran as unformatted sequential file
Contains record delimiters, usually 4 or 8-bytes at the beginning and end of each record
Corresponding C-code has to "decipher" these delimiters

Summary

With ISO_C_BINDING, Fortran has standardized mechanisms to access source code and libraries written in C, as well as define Fortran accessible from C

Interoperability should be used with well-defined interfaces
- Complicated structures or calling sequences are not recommended

Use of STREAM I/O is recommended if data is to be interoperated from C
Parallel programming with Fortran coarrays?

- Adds \textit{parallel processing} as a 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) approach
  - Unlike OpenMP, which operates over shared memory, the coarrays implement parallelism over “distributed shared memory”
  - This approach is potentially massively parallel
- Have been integrated into Fortran 2008 standard

Some history

- “Coarray Fortran” (formerly known as F--) was created by Bob Numrich and John Reid in the 1990s
- Reached the current form in 1998 as a simple extension to Fortran 95 for parallel processing
- Coarrays have been in production for many years but mainly on Cray hardware
- A set of core features are now part of the Fortran 2008 standard, development continues in the upcoming Fortran 2015 standard

Partitioned global address-space model

- Partitioned global address-space (PGAS) programming model: each PE can access its \textit{local} data and (a portion of) \textit{remote} i.e. other PE’s data with \textit{single-sided} put/get (=write/read) operations
Why bother?

- Real-world experiences of improved performance when replacing MPI communication kernels with PGAS communication
- For example: partial inclusion of coarrays to ECMWF’s IFS

Word of warning about the compiler support

- Still today, compiler support for the coarray syntax is a bit limited
  - Cray Compiler implements coarrays properly
    - automatic recognition; no flags needed; production-level performance
  - Intel Compiler supports coarrays but on top of MPI (compile with -coarray)
  - GNU compiler (5.1 and newer) supports building a coarray code upon an external library (OpenCoarrays)
  - G95 compiler features also partial support

Execution model

- Upon startup a coarrays program gets replicated into a number of copies called images (i.e. processes)
  - The number of images is usually decided at the execution time
- Each replica (image) runs asynchronously in a non-coupled way until program-controlled synchronization

FORTRAN COARRAYS: BASIC CONCEPTS

- Execution model
**Execution model**

- Image’s data are visible within the image only – except for data declared as special arrays: **coarrays**
- One-sided memory accesses enable movement of coarray data across different images of a program

```
program foo
integer :: a[*]
call mpi_init(rc)
...
end program
```

**Declaring coarrays**

An entirely new data structure, coarrays, become meaningful in parallel programming context, when their data are remotely accessible by its images

Accomplished through additional Fortran syntax for coarrays for Fortran arrays or scalars, for example

```
integer :: scalar[*]
real, dimension(64) :: vector[*]
```

- Declares a **scalar** with a local instance on every image
- Declares a **vector** with a local instance consisting of 64 elements on every image

**Time for “Hello World”**

```
program hello_world
implicit none
write(*,*) 'Hello world from ', this_image(), ' of ', num_images()
end program
```

This program is a trivially parallel i.e. each image does not explicitly share any data and runs seemingly independently

```
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 1 to num_images()
```
Declaring coarrays

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

```fortran
integer :: global(3)[*], local(3)
global(:) = this_image() * (/ 1, 2, 3 /) ! local initialization
local(:) = global(:)[1]  ! copy from image number 1 to every ! image
```

Synchronization

- We need to be careful when updating coarrays
  - Is the remote data we are copying valid i.e. up to date?
  - Could another image overwrite our data without notice?
  - How do we know if the remote update (fetch) is complete?
- Fortran provides synchronization statements, e.g. adds a barrier for synchronization of all images
  ```fortran
  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 global(:)[1]
  ```fortran
  global(:) = this_image() * (/ 1, 2, 3 /)
sync all
local(:) = global(:)[1]
  ```

Synchronization

- In this particular case – since only the image #1 is in a critical position, we could use an alternative, pairwise form of synchronization
  ```fortran
  global(:) = this_image() * (/ 1, 2, 3 /)
sync images(1)
local(:) = global(:)[1]
  ```
- sync images can also take a vector of image ranks with which to synchronize
  ```fortran
  sync images( (/1, 2/) )
  ```
**Case study: parallel sum of array elements**

- Array originally on image #1 (I1)
- Parallel algorithm
  - Scatter
    - Half of the array is read to image #2 (I2)
  - Compute
    - I1 & I2 sum independently their segments
  - Reduction
    - Partial sum on I2 written to I1
    - I1 sums the partial sums

**Step 1: scattering** - I2 fetches a lower part of the array

```fortran
integer, dimension(n) :: array[*]
integer :: psum[*]
...
if (this_image() == 2) then
  array(1:n/2) = array(n/2+1:n)[1]
end if
```

**Step 2: Both images compute local sums**

```fortran
psum = sum(array(1:n/2))
```

**Interim summary: basic concepts**

- Concept of images and some related functions
- How to declare “codimensional” arrays (coarrays) and access their data
- Image synchronization
Case study: parallel sum

Step 3: Image 1 reads the partial sum from Image 2 and computes the total sum

\[ \sum = \sum \]

\[
\text{sync all}
\]
\[
\text{if (this_image() == 1) then}
\]
\[
\quad t\_sum = p\_sum + p\_sum[2]
\]
\[
\text{end if}
\]

DYNAMIC MEMORY ALLOCATION FOR COARRAYS

Allocatable coarrays

- It is possible to define dynamic coarrays

  integer, allocatable :: a(:, :)

  ...

  allocate(a(1000)*)

- allocate and deallocate imply implicit synchronization — all images must participate, i.e., an implicit sync all occurs
- The local size must be the same on each image or otherwise a runtime error will occur
- The last codimension must have an asterisk “*”

About pointers with coarrays

- A coarray declaration cannot have the pointer attribute
- Thus the following definition is illegal:

  real, pointer :: ptr[] ! this is invalid Fortran

- However, we can define a new type, where type component(s) have pointer (or allocatable) attributes
  - And then define a coarray being of that type
- Used in dynamic dimensioning of coarrays
  - In this way, the local sizes on every image can be different
Variable length coarrays via allocatable

- Create a new Fortran data type with `allocatable` component in it – and place it in a module
  ```fortran
type mytype
    real, allocatable :: data(:)
end type mytype
```

- Then declare a coarray of that type, e.g.
  ```fortran
type(mytype) :: co[*]
```

- Allocate on each image, but *different size*
  ```fortran
allocate(co % data(10 * this_image()))
```

- Refer to the data of another image as, e.g.,
  ```fortran
element = co[1] % data(1)
```

Coarrays in procedures

- When declared in a subroutine or function, a coarray must be one of the following
  - Declared as a dummy argument to the procedure
  - Have `allocatable` and/or `save` attribute
- Re-mapping of corank is also allowed
- A coarray in procedure cannot be an automatic array

```fortran
subroutine some_routine(n, array_1, co_array_1, array_2, co_array_2, co_array_3)
! Procedure arguments
integer :: n
integer :: array_1(n), array_2(:) ! explicit and assumed shape
integer :: co_array_1(n)[*], co_array_2(:)[*] ! explicit and assumed shape
integer :: co_array_3::* ! illegal: assumed co-shape

! Procedure variable declarations
integer :: local_array_2(1000) ! local array
integer, save :: local_co_array_3(1000)[*] ! coarray with save attribute
integer :: local_co_array_2(1000)[*] ! illegal: coarray without save
integer, allocatable :: local_co_array_4(::* ! coarray with allocatable
integer :: local_co_array_1(n)[*] ! illegal: automatic co-arrays
integer, pointer :: local_co_array_5(:)[::] ! illegal: coarray with pointer
end subroutine some_routine
```
COARRAYS AND (FILE) I/O

I/O conventions

integer :: param[*] = 0
...
if ( this_image() == 1 ) then
  read *, param
  do i = 2, num_images()
    param[i] = param
  end do
end if
sync all

Do stdin with the image #1 only and broadcast the data

Parallel I/O with coarrays

Spokesman strategy

– One image takes care of all I/O using normal read/write routines
  ▪ Data gathered explicitly to the spokesman image
– Does not scale, single writer is a bottleneck
  ▪ Single I/O client not able to fully utilize a parallel filesystem
– Can be good option when the amount of data is small (e.g. input files, log files)

Coarrays and I/O

– Each image has its own, independent set of Fortran input and output units
– The default input (“stdin”, i.e. read(*,…) etc) is pre-connected to the master image (image #1) only
  – Do stdin with the image #1 only and broadcast the data
  – The same applies to command-line input
– The default output (“stdout”) is connected to all images
  – Output is merged (in no particular order) into one output stream
  – The standard error (“stderr”) is redirected to the “stdout”
Parallel I/O with coarrays

Every man for himself
- Each image writes its local results to a separate file
- Good bandwidth
- Difficult to handle a huge number of files in later analysis
- Can overwhelm filesystem (for example Lustre metadata)

Sketchy examples

Spokesman strategy
if (this_image() == 0) then
    open(unit=10, file="data.dat", form='unformatted', access="stream")
    do i = 1, num_images()
        write(10, positions="append") data[i]
    end do
    close(10)
end if

Every man for himself
write(filename,'(A5,I0,A4)') 'file.', this_image(), '.dat'
funit = 10+this_image()
open(funit, file=filename, form='unformatted', access='stream')
write(funit) data
close(funit)

Parallel I/O with coarrays

Subset of writers/readers
- Good compromise but more complex
- Number of readers/writers could be e.g. sqrt(num_images())
- If readers/writer images are dedicated, some computational resources will be wasted

Summary of the second part

How to define coarrays with varying-sized local parts
Using coarrays in procedures
I/O and Fortran coarrays
FURTHER TOPICS: MULTIPLE CODIMENSIONS

Multiple codimensions

Multi-dimensional coarrays are possible also in terms of codimension

– The last codimension must always be declared with asterisk "*

– The bounds of codimensions start from 1 by default but can be adjusted

```plaintext
integer, codimension[2,*] :: scalar
real, dimension(64,64), codimension[4,*] :: matrix
real, dimension(128,128,128), codimension[0:1,0:1,0:*] :: grid
```

Multiple codimensions and this_image

– In addition to returning the local image number, this_image can also return the cosubscripts of a coarray

An alternative form of the function is this_image(coarray[, dim])

where coarray is a coarray of any type.

– If dim is absent, the result is an array of rank one with the size equal to the corank of the coarray holding the cosubscripts of the image holding the coarray

– If dim is present, integer cosubscript of the coarray corresponding to dim is returned.

this_image: example

– The set of images fills the coarray contiguously starting from the first codimension

See the table for an example case where 10 images in total are used

<table>
<thead>
<tr>
<th>Image</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>this_image(a)</td>
<td>(0,3)</td>
<td>(1,3)</td>
<td>(2,3)</td>
<td>(3,3)</td>
<td>(0,4)</td>
<td>(1,4)</td>
<td>(2,4)</td>
<td>(3,4)</td>
<td>(0,5)</td>
<td>(1,5)</td>
</tr>
</tbody>
</table>

Cosubscripts of array a when running with 10 images in total
Occasionally it is useful to perform an opposite conversion of this image, i.e., to extract image index based on some given cosubscripts.

To get an image index based on the cosubscripts of a coarray, image_index intrinsic function can be used:

\[
\text{image_index}(\text{coarray}, \text{sub})
\]

- If \(\text{sub}\) holds valid cosubscripts for \(\text{coarray}\), the return value is an integer describing the corresponding image index.
- Otherwise the result is zero.

Obtaining lower and upper bounds for coarrays

There are query functions for obtaining upper/lower bounds:

\[
\begin{align*}
\text{lcobound} & (\text{coarray} [, \text{dim}]) \\
\text{ucobound} & (\text{coarray} [, \text{dim}])
\end{align*}
\]

They return an integer array, or a scalar in case of a single dimensional query (with optional “dim”).

For the array \(a\) from the previous page:

\[
\begin{align*}
\text{lcobound}(a) & \text{ is } 0 3 \\
\text{ucobound}(a, \text{dim}=1) & \text{ is } 3
\end{align*}
\]

Case study: parallel matrix-matrix multiply

Consider computing a blockwise matrix-matrix multiply:

\[
C = AB \iff C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}, \quad X = \begin{bmatrix}
X_{11} & \cdots & X_{1N} \\
\vdots & \ddots & \vdots \\
X_{N1} & \cdots & X_{NN}
\end{bmatrix}
\]

Image indexes returned by \(\text{image_index}([x,y])\) for array \(a\) when running with 10 images in total:

<table>
<thead>
<tr>
<th>[x, y]</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

If the given cosubscripts are valid, index of the image holding the data is returned. Zero return value indicates invalid cosubscripts.

See the table for an example case where 10 images in total are used.
Case study: parallel matrix-matrix multiply

Data distribution: image (i, j) holds data for blocks A_{ij}, B_{ij} and C_{ij}.

To compute C_{ij}, each image needs to remotely read A_{ik} and B_{kj}, for k = 1, \cdots, N, k \neq j from other images.

The block matrix-matrix products A_{ik}B_{kj}, for k = 1, \cdots, N readily computed with matmul intrinsic function.

ADVANCED SYNCHRONIZATION

<table>
<thead>
<tr>
<th>Images</th>
<th>Time (s)</th>
<th>S(1)/S(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>85.4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>24.7</td>
<td>3.5</td>
</tr>
<tr>
<td>9</td>
<td>13.6</td>
<td>6.3</td>
</tr>
<tr>
<td>16</td>
<td>8.2</td>
<td>10.3</td>
</tr>
</tbody>
</table>

Results for N=10000 on sisu.csc.fi

ADVANCED SYNCHRONIZATION
At times it is necessary to execute a block of code by one image at a time to avoid race conditions whilst updating coarrays at a fixed image location, e.g. [P] critical – end critical is the construct to use

Remember to synchronize especially after the section

type (lock_type) :: lockvar[*]

... if (this_image() == p) then
   lock(lockvar[q])
   p[q] = p[q] + 1
   unlock(lockvar[q])
   sync images (q)
end if

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

Enforces any pending coarray 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 coarray(s)

Not actual synchronization at all, however sync all / sync images also imply sync memory

UPCOMING COARRAY FEATURES

 syncing memory

 integer :: a[*]

if (this_image() == 1) then
  a[2] = 123  ! put 123 to image#2
  ! image#2 may not yet see the update
  sync memory
  ! image#2 now sees the update
end if
**Technical specification**

- Fortran, coarrays included, continue to evolve
- Technical specification (TS) outlines the future developments
- In the pipeline are the following enhancements
  - Image teams
  - Failed images
  - Events
  - Collective intrinsic functions for coarrays
  - Improvements in atomic updates

**TS: Image teams**

- Learning lessons from MPI and its communicators, it is useful to have subsets (teams) of images
- Define a new team at runtime
  - For example divide images into two teams
    
    \[
    \text{id} = \text{mod}(\text{this_image}(), 2) + 2 \\
    \text{form team (id, odd_even)}
    \]
- Synchronize within a team only
  
  \[
  \text{sync team (odd_even)}
  \]

**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
  
  \[
  \text{sync images (image_list, stat = st)} \\
  \text{if (st == stat\_failed\_image) then ...}
  \]
- `failed_images()` function returns the count of failed images at the given moment
  - Another possibility is to use `num_images(failed = .true.)`
- Together with the new team capability, fault-tolerant programming becomes possible (still non-trivial)

**TS: Collectives**

- Collective operations are essential part of parallel programs
- TS contain definitions for
  - `co.broadcast`
  - `co.sum`, `co.min`, `co.max`
  - `co.reduce`
- These are available already on the Cray compiler
  - Faster than corresponding explicit communication patterns
Summary of the last part

- Use of multiple codimensions sometimes handy
  - Using specific intrinsic functions `lcobound`, `ucobound`, `this_image` and `image_index` usually clarify the situation
- More advanced synchronization across images
- Future features: image teams, fault-tolerant utilities, collectives
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

```fortran
module types_and_procedure_pointers

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

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

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)

end module
```

Derived types: examples of declaration

```fortran
module derived_types

type base_type
  integer :: field = 1
end type base_type

type :: ...
end type array_type

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

end module
```

```fortran
module derived_types

type base_type
  integer :: field = 1
end type base_type

type :: ...
end type array_type

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

end module
```
Derived types: component visibility

- 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.
    - Also procedures with arguments of private type.

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.

Types: parameterization

- 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.
Types: parameterization

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

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

Derived types: parameterization example

```fortran
! Type definition
type matrix(rk, m, n)
  integer, kind :: rk
  integer, len :: m, n
  real(kind=rk) :: value(m,n)
end type matrix

! Declaration
integer, parameter :: &
dp = selected_real_kind(8)
type(matrix(dp,10,10)) :: m1

! Allocation
allocate(matrix(dp=dp,m=30,n=30)::m3)

! KIND and LEN inquiry
write (*,*) m1 % rk, m1 % m, m1 % n
```

Type initialization

```fortran
module mymod
  type person
    integer :: age = 0
    character(len=20) :: name
  end type
end module

program test
  use mymod
  type(person) :: p1, p2, p3
  p1 = person(50, 'Peter')
  p2 = person(age=43, name='Cindy')
  ! age field has default value,
  ! can omit keyword
  p3 = person(name='John')
end program test
```

Derived types: initialization example

```fortran
! Type definitions
type pointer_type
  type(body) :: data
  type(pointer_type), pointer :: next
end type

! Type allocations
type(body) :: data1, data2
type(pointer_type), target :: r1
type(pointer_type), target :: r2
type(alloca_type) :: a1, a2

! Pointer components
r2 = pointer_type(data2, null())
r1 = pointer_type(data1, r2)

! Allocatable components
a1 = alloc_type('null',null())
a2 = alloc_type('vec',[1.2,2.3,4.5])
```
Derived types: assignment semantics

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

```
type(pointer_type) :: r1, r2
r1 = r2 ! Equivalent to
    ! r1 % data = r2 % data
    ! r1 % next => r2 % next
```

```
type(alloc_type) :: a1, a2
a1 = a2 ! Equivalent to
    ! a1 % type = a2 % type
    ! if (allocated(a1 % data)) deallocate(a1 % data)
    ! allocate(a1 % data(size(a2 % data)))
    ! a1 % data = a2 % data
```

ASSOCIATE construct

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

```
associate (association_list)
block
end associate
```

List of associations is of the form association:
- `associate_name => variable`
- `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, \texttt{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 \texttt{associate\_name} is a variable, it can be used as a variable within the block and is assignable.
If \texttt{associate\_name} is an expression, association is used for the expression value within the block and is not assignable.

Generic procedures

Procedures which perform similar actions but for different data types can be defined as \texttt{generic procedures}.

Procedures are called using the \texttt{generic name} and compiler uses the correct procedure based on the argument number, type and dimensions.
- Compare with “overloading” in C++.

Generic name is defined in \texttt{INTERFACE} section.
Generic procedures example

```fortran
module swapmod
  implicit none
  interface swap
    module procedure swap_real, swap_char
  end interface
contains
  subroutine swap_real(a, b)
    real, intent(inout) :: a, b
    real :: temp
    temp = a; a = b; b = temp
  end subroutine
  subroutine swap_char(a, b)
    character, intent(inout) :: a, b
    character :: temp
    temp = a; a = b; b = temp
  end subroutine
end module
```

Operator overloading

Operators can be overloaded with the same approach.

Instead of procedure name, use operator keyword, for example:

```fortran
interface operator(+)
  module procedure add_my_types
end interface operator
```

Interfaces

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.

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

Abstract interface example

```
! Abstract interface definition
abstract interface
  subroutine sub_with_no_args()
  end subroutine sub_with_no_args
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
procedure(real) :: y
```

PROCEDURE statement

- Abstract interface can be referenced with procedure statement:
  ```
  procedure([iface]), attrs :: decl-list
  ```
  where `attrs` 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.

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.
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:
  ```fortran
  import [ [::] import-name-list]
  where import-name is an entity to be accessed from the host
  import without arguments makes all host entities accessible
  ```

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 :: p
p => mysin
write (*) p(1.0) ! Prints mysin(1.0)

type(base)
procedure(sub), pointer, pass(x) :: p
end type base

abstract interface
  subroutine sub(a, b, x)
  import base
  real :: a, b
  type(base) :: x
end subroutine sub
end interface

type(base) :: t1
t1 % p => mysub
! Equivalent to call mysub(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

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
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`.

```fortran
type :: base
  integer :: field1
end type base

type, extends(base) :: extended
  integer :: field2
end type extended
```

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.

All components as well as type parameters of the `parent type` are inherited by `extended type`.

Overriding of old type-bound procedures is allowed.

Extended type has an additional component of the parent type with the name `parent type`.

Type extension example

```fortran
! 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

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

type :: base
  integer :: field1
end type base

type, extends(base) :: extended
  integer :: field2
end type extended

type(extended), target :: t1
class(base), pointer :: p1

t1 = extended(1,2) ! p1 is of class base
p1 => t1

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 is either pointer or allocatable
– Unless variable is used as a dummy argument

Polymorphic variables are type-compatible with type_name and all extensions of type_name

Polymorphism example

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'

Polymorphism limitations

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

There are also limitations for performing input/output with polymorphic variables
POLYMORPHIC 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.

Sourced object initialization

Values may be initialized based on a SOURCE object.
allocate(aloc, SOURCE=expr)
SOURCE object expression expr has to be type-compatible with alloc. For arrays, expr and alloc must be of the same rank.

Polymorphic object initialization

ALLOCATE statement is extended to allow definition of type, type parameter values, type values and size (for array objects).
For example:
class(base_type), allocatable :: a
can be allocated with
allocate(type(extended_type)::a)
Sourced object initialization example

```fortran
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 example

```fortran
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 ''
```

For Fortran 2008 allows objects to have the shape, type and type parameters without copying the object values by using **MOLD**

```fortran
allocate(aloc, MOLD=expr)
```

**MOLD** object has same restrictions as **SOURCE**

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

**SELECT TYPE**
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.

subroutine write_output(obj)
  implicit none
  class(base) :: obj

  select type(obj)
    type is (extended)
      write (*,*) obj % field1, &
      & obj % field2
    type is (base)
      write (*,*) obj % field1
  end select
end subroutine write_output

SELECT TYPE construct cont.

Each guard_stmt is one of
- type is (type_spec)
- type is (intrinsic_type [(param_value_list)])
- class is (type_spec)
- class default

with type specifier type_spec is defined before.

type_spec is required to be type compatible with selector.

Intrinsic type specifiers are allowed only when selector is unlimited polymorphic object (class(*)).

SELECT TYPE second example

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

Type of polymorphic objects may be determined at runtime by using SELECT TYPE construct.

select type ([assoc_name =>] selector)
  [guard-stmt [name]
   block]
...end select

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.

**Unlimited polymorphic entities**

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

**PROCEDURES AS TYPE COMPONENTS**
Procedures as type components

- In OO programming, procedures acting on the data are bound 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 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`
- `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_employee =>
  print_emp
  per % print_person()
  call emp % print_employee()`
Type-bound procedures example cont.

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

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

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

type :: employee
integer :: salary
contains
  procedure, non_overridable :: &
  print_employee => print_emp
end type employee

type(person) :: per
name = 'Joe', age = 10
per = person(name=%'Mike', age=42, &
salary=2000)

! Calls print_per
! Calls print_info()
```

Procedure overriding

- A way to 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 overridden simply defining a new procedure with the same name.
- Overriding procedure must have **exactly** the same interface as the overridden procedure.
  - Apart from the type of the passed-object dummy argument which **must** be of the extended type.
- **NON_OVERRIDABLE** attribute to prevent overriding.
Summary

In the object-oriented programming paradigm data is coupled with the methods operating on the data.

Fortran 2003/2008 supports the object-oriented programming model with:
- Type extensions
- Type-bound procedures
- SELECT TYPE construct for determining the type during runtime.
Type component access control in Fortran 2003

- Fortran 2003 allows mixed component accessibility for derived types
- 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

Type access control example

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


generic interfaces and procedures

- 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.
  
  ```fortran
  interface [name/operator/]=] [module procedure proc_name_list]
      [module procedure proc_name_list]
  end interface
  
  Visibility of the generic `interface` is always `public`

GENERIC type-bound procedures

- Type-bound procedures may be defined as generic with the `GENERIC` statement.
  ```fortran
  generic :: 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(mynname, &
    myage) result(per)
  character(len=10) :: myname
  integer :: myage
  type(person) :: per
  per = person(name=mynname, age=myage)
end function create_person

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 module person_module
```
**GENERIC procedures example cont.**

```fortran
program example
  use person_module
  implicit none
  type(person) :: t1
  character(len=10) :: name
  integer :: age
  
  t1 = person('Joe', 10) ! Invokes create_person
  call t1 % get(name)   ! Invokes get_name
  call t1 % get(age)    ! Invokes get_age
end program example
```

**ABSTRACT TYPES**

**Abstract types**

```
type, abstract :: abs_base
end type abs_base

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

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

t1 = extended(1)
p1 => t1
```

- 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

- An type can be **abstract** with **deferred** procedures having no actual implementation
  ```fortran
  type, abstract :: type_name
  ...
  contains
    procedure(iface), deferred :: tbpname
  ...
end type type_name
  ```

- **deferred** procedures can only be used with **abstract** types
Abstract types cont.

- 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 example

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

Abstract types example cont.

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

FINAL COMPONENT
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.**

```
! Data containing handle
contains
  final :: close_handle
end type handle
```

```
subroutine close_handle(this)
  implicit none
class(handle) :: this
! Implementation omitted
end subroutine close_handle
```

Subroutines of a derived type can be declared final:

```
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.

```
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