Log in to the LRZ Linux Cluster
The windows command line (Start --> Enter command) should be used to execute the following two commands:

• XMing – multiwindow
• PuTTy

In the PuTTy window, please enter the node name “lxlogin10.lrz.de” to access the system, using the account assigned to you, and the password provided on the blackboard. Once you have logged in, you can start additional programs like xterm or editors like vi or emacs.

Compilers
The following compilers are available for these exercises:
• gfortran (part of GCC, the GNU compiler collection)
• ifort (Intel Fortran compiler – commercial);
• nagfor (NAG Fortran compiler - commercial)

By default, the Intel compiler is configured for ready use. For other compilers, you need to switch to the appropriate environment by executing:

```
module unload mpi.intel intel
module load gcc/8          (for use of gfortran) or
module load nag            (for use of nagfor) or
module load intel/18.0     (for use of a more recent ifort version)
```

The corresponding compiler commands are: ifort, gfortran, nagfor, respectively.

Note that after switching you will need to rebuild previously generated module information files as well as object files. Setting up the environment for coarray programs requires further preparations (in addition to those above):

```
module load mpi.intel/2018_gcc caf/gfortran     (for use of gfortran CAF) or
module load mpi.intel/2018 caf/intel           (for use of ifort CAF)
```

The command to build coarray programs with gfortran is caf, while for Intel Fortran you should use the mpif90 command and add the compiler switch –coarray.

Setup for performance measurements and parallel programs
Especially for the coarray Fortran programs (exercises from end of day 4 and day 5), it may be more advantageous to use a dedicated system. For this purpose, we will use dedicated nodes from the “ivymuc” Linux
cluster segment. Please use the following procedure (on a separate terminal/shell) to execute a parallel program on that system:

```bash
module load salloc_conf/ivymuc
salloc --reservation=fortran_course --cpus-per-task=2
cafrun -n 8 ./mycaf.exe # for gfortran
export FOR_COARRAY_NUM_IMAGES=8; ./mycaf.exe # for ifort
```

### Debuggers / Tools

The following debuggers and tools are available:

- **gdb** (GNU debugger) and **gdb-ia** (Intel-modified version of GNU debugger)
- **Valgrind** (memory debugger – `module load valgrind`)
- on the LRZ cluster: **PLplot** (`module load plplot`), **GSL** (`module load gsl fgsl`)

**Note:** commercial products may be only used during the course.

### Editing sources and makefiles

For editing Fortran source files or Makefiles, using either the **vi** or **emacs** editor is recommended. However, other preinstalled editors (e.g. **kate**) may also be used.

### Fortran standard document

A draft of the Fortran 2008 standard is available in PDF format at

http://www.j3-fortran.org/doc/year/10/10-007.pdf

It is recommended to use this document for reference, e.g., looking up standard intrinsic procedures (these are in section 13.7) as well as syntax definitions. As far as the general language rules are concerned, the document may turn out to be rather hard to read and therefore unsuitable as a guide for learning the language semantics.

### Further readings and information about the language

For quick reference on selected language elements you may also consult the documentation provided by the compiler vendors, e.g. here for Intel and GNU, but be aware also of non-portable vendor extensions:


https://gcc.gnu.org/onlinedocs/gfortran/

### Further notes on the exercises

- There is **no need to do all** of them.
- You are bound to make mistakes. Therefore, writing test programs (comparing expected with actual results) is a vitally important step in the development cycle. And one learns more from the mistakes than from the successes.
- Source code including **examples** from the talk and **solutions for the exercises** will be provided at the end of each day via the download URL:
Evaluation procedure
At the end of the course, you will be asked to fill in the electronic evaluation form at https://events.prace-ri.eu/event/759/evaluation/evaluate before you receive your course certificate.
Exercises for Day 1

Session 1 – Reverse communication interface

Some knowledge of OpenMP is useful for doing this exercise.

The directory `hands_on/skel/reverse_communication` contains example code for the integration routine that uses reverse communication, as discussed in the slides.

a. Why is that code (still) not thread-safe? You can check the fact by activating the OpenMP clauses in the calling program and running with OMP_NUM_THREADS set to a value larger than 1.

b. Re-design the code to be thread-safe without adding any OpenMP directives in the module `mod_integration`. Hint: Introduce a derived type.

Session 1 – Dynamic memory management for the heat example

The directory `hands_on/skel/heat` contains source code for the heat conduction example from the slide talk. This version of the program is rather inflexible, because you need to recompile it if you want to run a different problem size. Make changes to the source files that allow you to determine the problem size at run time by reading the array dimensions `nx` and `ny` from standard input via the `GET_COMMAND_ARGUMENT` and `GET_COMMAND_ARGUMENT_COUNT` intrinsics.

Note: A simple example for how to parse command arguments is available in the folder `examples/day1/command_line`. Also, you should consult the Intel and the gfortran compiler documentation for complete information on how to use the intrinsics.

The executable should then be invoked e.g., via

```
./heat.exe -nx 50 -ny 100
```

to a problem size 50 x 100. The following steps will be also needed:

- Make the necessary changes to the declarations of the fields `phi` and `phinew`. What else is needed to assure the fields are set up correctly?
- Note that the x and y directions now may be differently discretized, and the space differentials must be set up at run time.

The correctness can be checked by running a small problem size (10 x 10) to conclude fast, and activating a printout done while the iteration still executes. When you are done, perform comparative performance measurements for the problem size 200 x 200. Note that adding suitable vectorization options can significantly improve performance.

The solution for this exercise will be in the folder `hands_on/solutions/heat`, in the files `mod_heat.f90` and `heat.f90`. 
Session 2 – some performance tunings for the heat example

Return to the heat conduction example from session 1 and do some performance evaluations.

1. First, add the TARGET attribute to the phi and phinew arrays and observe the impact on performance for \( nx = ny = 200 \).
2. Due to copying of phinew to phi, a factor of two more memory accesses are needed than would be the case if the fields had the POINTER attribute and the pointers are simply switched between iterations. Make a copy of the module within which just that is done and check the performance (possibly for different compilers).
3. Does adding the CONTIGUOUS attribute to the pointer fields improve performance? If not, consider possibilities to eliminate the POINTER attribute for the duration of the calculations.

The solution for this exercise will be in the folder hands_on/solutions/heat, in the files mod_heat_ptr.f90 and heat_ptr.f90.

Session 2 – Operations on sparse matrices

The derived type sparse mentioned in the slides can be used to implement sparse matrices, and thus a sparse matrix-vector multiplication. A sparse matrix is represented by a rank-1 array

\[
\text{type( sparse ), allocatable :: sa(:)}
\]

which is allocated and subsequently has its (few) matrix elements set via repeated invocation of a module procedure set_element() on each array element of sa (the latter is intended to correspond to a matrix row in form of a linked list ordered by ascending index values). The folder hands_on/skel/sparse contains files mod_sparse.f90 and test_sparse.f90 copies of which you will need to suitably modify:

1. Add a module function that implements the multiplication of a sparse matrix by a real(dk) vector and overload the multiplication operator.
2. Run the resulting program and check the results. Then, check for memory leaks by building with debugging options and running the command through Valgrind:

   \[
   \text{valgrind --tool=memcheck --leak-check=yes --track-origins=yes --show-reachable=yes ./test_my_sparse.exe}
   \]

   Where does the observed problem come from?
3. Consider further changes to your program to avoid the problem.

Two slightly different solutions are provided in the hands_on/solutions/sparse folder as

- mod_sparse_simple.f90 and test_sparse_simple.f90
- mod_sparse.f90 and test_sparse.f90

Note: the sparse implementation demonstrated here is horribly inefficient, because pointer chasing causes loss of spatial locality. A number of much better-performing storage schemes for sparse matrices exist, which you should use in productive environments.
Session 3 – Sparse matrices continued

In the slides for session 2, an overloaded structure constructor was shown for an object of type `sparse`. This suggests a much more efficient way of storing the row of a sparse matrix. Create a new folder, copy the module to it and improve the type definition. Apart from adding a constructor overload, you will also need to modify the implementations of the module procedures, of course. Then check that the main program still executes with the same results without any changes to it. Finally, create a new main program that uses the overloaded constructor instead of setting elements individually. The solution to this exercise will be contained in `hands_on/solutions/sparse_crs`.

Session 4 – Physical bodies

Starting from the type definition

```fortran
  type :: body
    real(dk) :: mass
    real(dk) :: pos(3), vel(3)
  end type
```

please define two extensions of that type (in the same module that defines the above type) which describe

- a body that has an electrical charge,
- a body that rotates.

Then write a procedure `print_body` that prints out all components of an object whose dynamic type may be any of the three types described above.

In a main program, please declare an allocatable polymorphic variable of declared type `body`, allocate it to be in turn any of the three types above, define reasonable values, and invoke `print_body` on that object.

Suppose a physical body starts out being of `type(body)` but later acquires a charge (for example, by being struck by lightning). How can the previous values of the non-charge type components be retained when the object changes its dynamic type?

Finally, declare an unlimited polymorphic entity in the main program and allocate it to be of one of the above types. How can you invoke `print_body` on that object?

You can start out from the skeleton code given in `hands_on/skel/polymorphic_body`. The solution will be available in `hands_on/solutions/polymorphic_body`.

Session 4 – Implement the date class (part 1)

Fill in all necessary details needed for implementation of the date and datetime types discussed in the lecture. Some subroutines which do the necessary computational stuff are provided in `hands_on/skel/datetime` (please ignore the file `person.f90` for now). Please start out by implementing the missing subroutines `inc_date` and `inc_datetime` as well as overloading the structure constructor for the above types.
Exercises for Day 2

Session 5 – Implement the date class (part 2)
Bind the procedures written in part 1 of this exercise as well as write_date to their appropriate types and uncomment the statements in the test program that execute them, to check that your code works correctly. The solution for this is in the folder hands_on/solutions/datetime.

Session 5 – Lazy deletion of function values
Suppose you have a type definition

```fortran
module person
  private
  character(len=nmx) :: name
  class(date), pointer :: birthday => null()
end module
```

and functions that initialize objects of that type, or return the name or birthday of a person. How can you print out the birthday of a person without needing to copy an object? The datetime skeleton from the exercise above also contains a file person.f90 that you can start out from.

The solution for this is in the folder hands_on/solutions/datetime. Note: Because many compilers mishandle this, please check against the result produced by the NAG compiler.

Session 5 – Geometric objects
A rectangle might be characterized by the type definition

```fortran
module rectangle
  private
  real(dk) :: length = 0.0_dk
  real(dk) :: breadth = 0.0_dk
end module
```

Implement a type-bound procedure which calculates the area of a rectangle, as well as an initialization procedure and a procedure to perform a stretching of a rectangle along a chosen direction. Since a square is a special kind of rectangle, one might consider the following type definition:

```fortran
module square
  extends(rectangle)
  ! fill in missing bits
end module
```

How do you handle the inheritance behavior, especially that of the type-bound procedures? Assume that a function adjust exists that takes an object of type(rectangle) as an argument and increases the size of the object if its area is too small. What happens if an entity of type(square) is used as an actual argument in an invocation of this function? Try to improve on the type design.

Please consult the two Fortran codes in hands_on/solutions/shapes for the “bad” and “good” versions of the design.
Session 6 – improving an interface class
The folder examples/day2/interface_class contains a runnable example for the interface class discussed in the lecture. Unfortunately, the main program still contains a dependency on the module that declares the type extension. What is the cause of the problem? Try to find a solution such that the statement “use mod_file_handle” can be replaced by “use mod_handle”.

The solution will be available in hands_on/solutions/interface_class.

Session 6 – Collision of physical bodies
Write a type-bound procedure that calculates the front-on collision of physical bodies, starting from the code written for the above exercise. The following assumptions may be made: For the base type, the conservation of energy and momentum defines the result. In the center-of-mass system that moves with velocity

$$\vec{V} = (m_1 \vec{v}_1 + m_2 \vec{v}_2)/(m_1 + m_2)$$

the velocities $$\vec{w}_i' = \vec{v}_i' - \vec{V}$$ after the collision can be obtained from the corresponding $$\vec{w}_i$$ before the collision by

$$m_1 \vec{w}_1' + m_2 \vec{w}_2' = 0 = m_1 \vec{w}_1 + m_2 \vec{w}_2 \quad m_1 w_1^2 + m_2 w_2^2 = m_1 w_1^2' + m_2 w_2^2'$$

For two charged bodies that collide the charges after the collision are defined by

$$\frac{q_1'}{R_1} = \frac{q_2'}{R_2}$$

and you can assume that the electrostatic interaction is negligible. The radii $$R_1, R_2$$ of the objects shall be supplied through the interface.

Session 7 – Using asynchronous I/O
The directory hands_on/skel/aio contains source code for a ray tracer. This code performs I/O of the complete picture array at the end of the calculation. The resulting file can be viewed with the display command. Convert this program to use asynchronous I/O by putting the data transfer statements inside the outer loop that processes the tiles in the picture. In particular, you can reduce the needed amount of storage from $$\text{size}^2$$ to $$\text{size} \times \text{nbuf}$$, where $$\text{nbuf}$$ is the number of I/O buffers available for asynchronous I/O. For which picture size do you observe a performance advantage? For good I/O performance it is recommended to change to the $SCRATCH directory and execute the program there.

Session 7 – User-defined derived type I/O for sparse matrices
Starting from the solutions for the sparse matrix multiplication from session 2, add a generic binding and module procedures that support formatted as well as list-directed output of objects of type sparse. Modify the test program to exercise these features.

The solution for this exercise is contained in the folder hands_on/solutions/sparse (files mod_sparse_io.f90 and test_sparse_io.f90).
Exercises for Day 3

Session 8 – Using IEEE exceptions

Direct evaluation of the expression $|x| = \sqrt{\sum x_i^2}$ (euclidean norm of a vector) may cause avoidable overflows. Use the IEEE overflow flag to deal with this situation in a function subprogram that evaluates the above expression. Please test your code with at least the following input:

$$v = [2.0, 2.0, 2.0, 2.0]$$
$$v = [0.0, 0.0, \sqrt{2.0}, \sqrt{2.0}]$$
$$v = [0.0, 0.0, 0.0, 0.0]$$
$$v = [0.0, 0.0, 0.0, 2.0*\text{tiny}(1.0)]$$
$$v = [0.0, 0.0, 0.0, \text{tiny}(1.0)]$$
$$v = [0.0, 0.0, \text{tiny}(1.0), \text{tiny}(1.0)]$$
$$v = [2.0, 2.0, 2.0, \sqrt{\text{huge}(1.0)}]$$
$$v = [2.0, 2.0, 2.0, 1.1*\sqrt{\text{huge}(1.0)}]$$

The solution will be provided in hands_on/solutions/ieee_exceptions.

Session 9 – Parameterized derived types

Take a look at the skeleton code you find inside hands_on/skel/pdt/.

1. As indicated, define an abstract parameterized derived type matT with kind parameter k and len parameters m, n. Then, define a type extension with a rank-2-real-array component mat. In the main program, declare allocatable objects of the kind indicated in the skeleton main program. Further, define an assignment that can handle different kind-parameters for rmatT, e.g. real32 and real64 and any len parameter. Also define a matrix multiplication that can handle different kind-parameters for rmatT, e.g. real32 and real64. Try to get the indicated code fragments in the main program running.

2. Now add the private attribute to the parameterized type rmatT. What is required to get the main program to compile and run again?

Session 10 (easy) – Calling GSL from Fortran

Write a program that calculates the value of the error function

$$\text{erf}(x) := \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

for values of the argument between 0 and 2. Use the implementation supplied by the GNU scientific library, and use the variant of the function that also calculates the error estimate. Documentation is available at

https://www.gnu.org/software/gsl/manual/
and the environment module gsl provides the necessary variables ($GSL_LIB) to link against the library. The solution for this exercise will be contained in hands_on/solutions/interop_gsl

Note that a nearly complete Fortran interface to the library is available via https://github.com/reinh-bader/fgsl/

Session 10 (more difficult) – Using a POSIX call from Fortran
The API call getpwnam() allows to extract information about a specific user name from the authentication databases. Write a Fortran module and program that contains the necessary type definition, BIND(C) interfaces and module procedures to produce and print out a user’s HOME directory. What must be taken care of in case a user name does not exist? Study the man page for getpwnam() for information on its interface.

Skeleton code for the exercise is in hands_on/skel/interop_struct. A fully working main program is available, but the module mod_passwd must be implemented.

The solution for this exercise will be found in hands_on/solutions/interop_struct.

Session 11 – Dynamic dummy arguments
In the lecture, we’ve seen how C-defined types with pointer components can be handled in Fortran 2003. Now consider the reverse: How can the Fortran interface subroutine generate_data(arr)
   real(c_float), allocatable, intent(out) :: arr(:)
end subroutine
- which is non-interoperable (why?) - be used from C? Assume that the data for arr is generated within the subroutine and that these data should be accessible within C. See hands_on/skel/interop_deferred_shape for skeleton code.

With TS 29113 (extensions to C interoperability, currently only supported by the Intel compiler), it is permitted to directly use interfaces such as the above, i.e. it is possible to add BIND(C) to the above interface. This removes the need to introduce a “handle” derived type. Implement a C main program that uses the TS’ facilities to do such a call after correctly setting up a descriptor for the allocatable object.

Session 12 – C library interface with function argument
The folder hands_on/skel/interop_interface contains source files c_libcall.[c,h] implementing a library call with prototype
float Sum_Fun( float (*fun)(float x, void *params), void *params );
which takes a function argument to be provided by the client. Furthermore, the following files are supplied:

- an example main program in C (cmain.exe) which you can build using the Makefile – you'll need to inspect this to see how the argument function is implemented.
- skeletons for a Fortran library module f_lib.f90 and a Fortran main program fmain.f90, which will not successfully compile ...

Here’s what you are required to do:
1. Please add the missing functionality in the Fortran code using only Fortran 2003-style interoperability features, so it performs the same function as the C code. What limitations apply?

2. Using the additional features from the interop TS 29113 (currently only supported in the Intel compiler), extend the semantics of the interface so that also non-interoperable arguments can be used from Fortran. Check that this works by calling the function with a polymorphic actual argument.

The solutions will be provided in `hands_on/solutions/interop_interface`. 
Exercises for Day 4

Session 13 – Trajectories of Physical Bodies

Define a type that – reusing code from the Session 4 exercise “Physical Bodies” - is capable of describing the trajectory of a particle of class(body). This means that the physical state of the particle at some number of points in time must be stored inside an entity of such a type.

1. What type components must be defined to fully describe such a trajectory? Write an overloaded (generic) structure constructor for an entity of that type that defines an initial condition for the trajectory.

2. Write a procedure that updates the trajectory by adding an additional time step, assuming that mass or charge of the particle do not change. Deal with the storage limitation of an entity of type(trajectory) by writing data to disk if necessary, without changing existing interfaces.

3. What do you need to do if an entity of type(trajectory) goes out of scope or is deallocated?

Please start out with the skeleton code in hands_on/skel/trajectory. The entries marked FIXME indicate where missing bits need to be added; a test program that runs the code is also supplied. Reference output is provided in a subdirectory.

Session 14 – Integration example

Fill in the details on implementing the integration example from today's lecture. Some code which provides a simple default integrator is available in hands_on/skel/integration, as well as code that indicates how Fortran GSL interface functions (which provide more refined capabilities) should be used. Please note that the main program needs to change (read the FIXME entries there), and the qdr.f90 module needs appropriate updates.

Session 15 – Reverting an object to its earlier state

Define a type and associated type-bound procedures that can be used for an object that is intended to describe the solution of a system of linear equations, \( Ax = b \). The first step used should be invocation of the LAPACK routine SGESV. If this routine fails, it will be necessary to restore the matrix \( A \) to its earlier state before calling a procedure that calculates the singular value decomposition (in which case one can at best obtain a least squares deviate). Implement the Memento pattern necessary to perform this restoration in a well-encapsulated way. The skeleton code you can start out from for this exercise is in the folder hands_on/skel/linear_solution_space.
Session 16 – Implementing the observer simulation
Starting from the skeleton code in `hands_on/skel/observer`, complete the implementation of the observer simulation. The energy value and plotting slices should be updated in intervals corresponding to a noticeable change in the function \( y(x, t) \); corresponding values of the simulation time should also be printed out. The skeleton code for this is in `hands_on/skel/observer`. Reference printout is also provided as `stdout.res`.

Session 17 – Data distribution
Please read the web page at [http://www.lrz.de/services/software/parallel/pgas/caf/](http://www.lrz.de/services/software/parallel/pgas/caf/) to inform yourself about how parallel Fortran programs are compiled and started. Then, consider a triangular matrix:

\[
A(i, j) = \begin{cases} i + j & \text{if } i = 1, j = 1 \ldots n-i+1 \\
\text{typically } n >> \text{number of tasks} & \end{cases}
\]

Make a copy of the serial program `hands_on/skel/triangle/triangular.f90` into your working directory. The program reads in matrix size and a row index from the command line. It then sets up

\[ A(i, j) = i + j \]

and prints out the specified row. Parallelize this program in a manner that distributes data as evenly as possible across images.
Exercises for Day 5

Session 18 – Heat conduction parallelized
Starting out from the skeletons `mod_heat.f90` and `heat.f90` introduced on day 1, parallelize this code using coarrays.

a. Introduce a one-dimensional domain decomposition along the y direction. One method to deal with the boundary cell problem consists in assigning each image an additional column (“halo” or “ghost” cells, colored grey in the figure below) at its boundaries to another domain which receives data from the task that hosts that domain:

Only the halo cells need to be involved in communication; in this example, these form contiguous arrays. For now, please only run a fixed (sufficiently large) number of iterations, omitting the termination criterion.

b. Start with a small problem size to check whether correct results (the printout from multiple tasks should be collected on image 1) are obtained.

c. When running with a problem size of 200 x 200, up to how many images does your code scale?

The solutions for this problem will be available in the folder `hands_on/solutions/heat_caf` (files `mod_heat_static.f90` and `heat_static.f90`).

Session 19 (alternative 1) – triangular matrix-vector multiplication
Continue the session 17 exercise by implementing the full matrix-vector multiplication for triangular matrices. Use the most efficient access pattern for calculation of the scalar products. The coarray used to store one of the vectors should be allocatable. Do not forget to implement the necessary communication to assure the complete result is available on all images. The solutions for this problem will be available in the folder `hands_on/solutions/triangle_full`. 
Session 19 (alternative 2) – parallelizing a ray tracer
The subdirectory hands_on/skel/ray contains a serial ray-tracer code (the same we've seen before in a different context), which computes a pretty picture. The central function is calc_tile(), which computes one tile of the picture. The size of one tile and of the whole picture is hardcoded at the start of the main program. Note that the code assumes that the picture size is a multiple of the tile size. In the version given, the picture size is 4000 x 4000 and the tile size is 200 x 200.

Parallelize the code using coarrays. You can deactivate the output for testing, but make sure that your parallel code computes the correct result (this is easy since you can always display the picture). What speedup does your code obtain going from 1 to 16 images? Also, compare with the baseline performance from the serial code.

Session 20 – Completing the heat example
Complete the parallel version of the heat conduction program by adding the necessary reduction call that enables consistent evaluation of the termination condition. Note that currently only gfortran supports the collectives defined in TS18508. For Intel Fortran, either a critical region must be used, or the Fortran collective must be replaced by an MPI call.

Session 21 (optional) – parallel library call
For the matrix-vector multiply, rewrite the main program to use a library call with a coarray dummy argument.
Table of Contents for the slides

Slide set days 1-3

Introductory remarks
- Overview of topics, conventions used, references: 2-7

Recapitulation: Module features and the environment problem
- Modules, global variables, encapsulation, module procedures, procedure arguments: 9-15
- Threading issues for globals: 16-18
- The environment problem and some of its solutions, particularly reverse communication: 19-30

Dynamic memory and object-based design
- POINTER and ALLOCATABLE attributes and their usage; some newer features: 32-35
- “Container types” – dynamic type components: 36-39
- Dynamic objects in procedure interfaces, semantics of INTENT: 40-46
- Opinionated recommendation on use of dynamic objects: 47
- Recapitulation of generic interfaces and overloading: 48-49
- A type representing sparse matrices; overloading the structure constructor and the assignment operator; finalizers: 50-56
- Aliasing rules for dynamic objects and TARGETs: 57-66
- ASSOCIATE construct: 67-68
- Library design and the Open Closed Principle: 69-71

Object oriented programming (I)
- General aims and scope of OO in Fortran: 73-74
- Type extension, inheritance mechanisms, component accessibility, structure constructor: 75-82
- Polymorphism for data and interfaces, polymorphic arrays, allocation mechanisms: 83-88
- Type and Class identification at run time (SELECT TYPE): 89-92
- Unlimited polymorphic objects, ad-hoc type comparison: 93-97

Object oriented programming (II)
- Type-bound and object-bound procedures, pointers to procedures, invocation of such procedures: 99-106
- Overriding a type-bound procedure, private type-bound procedures, the NON_OVERRIDABLE attribute, use of SELECT TYPE vs type-bound procedures: 107-112
- Passed object: arrays, the PASS and NOPASS attributes: 113-114
- “Interface classes” – abstract types and deferred type-bound procedures: 115-119
- Dependency inversion with submodules – splitting interfaces and implementations: 121-129
- Generic type-bound procedures, additional overloading features: 131-139

Advanced I/O Topics
- Nonadvancing I/O: 141-143
- User-defined derived type I/O: 145-151
- Asynchronous I/O: 153-159
- Performance and I/O design considerations: 161-166
IEEE Arithmetic and Exception Handling
- IEC 754 Standard, intrinsic modules for IEEE features: 168-171
- Floating point model, inquiry intrinsics for model parameters: 172-175
- Rounding and rounding modes, denormal numbers, inquiry of support for IEEE conforming operations, underflow handling: 176-183
- The five floating point exceptions, example on divide_by_zero: 184-187
- Halting mode, saving and restoring flag and floating point status: 188-191
- Classification of IEEE conforming number values, final comments: 192-194

Parameterized derived types (PDT)
- KIND and LEN parameters, comparing PDT with conventional container types, type parameter inquiry: 196-199
- Assumed, deferred and default type parameters: 200-202
- PDTs and object orientation: 203

Interoperation of Fortran with C in Fortran 2003
- Overview of Fortran 2003 interoperability, concept of companion processor: 205-207
- Interoperable intrinsic types and their use in interfaces, mapping of procedures, linkage: 208-211
- Passing simple arrays between Fortran and C: 212-215
- String handling, interoperable derived types: 216-219
- Handling non-interoperable types – the c_ptr type and the intrinsics c_loc and c_f_pointer: 220-230
- Procedure arguments and pointers – the c_funptr type and the intrinsics c_funloc and c_f_procpointer: 231-235
- Interoperation of global variables and COMMON blocks, enumeration: 236-238

Extended interoperation defined in Fortran 2015 (TS 29113)
- Overview of Fortran 2015 interoperability: 239
- The source file ISO_Fortran_binding.h – the C descriptor for “complex” Fortran dummy objects: 240-242
- Processing array elements in C: 243-245
- Creating Fortran objects within C: assumed shape, ALLOCATABLE, POINTER: 246-252
- Creating subobjects of strings or derived-type objects: 253-254
- Assumed-rank and assumed-type objects: 255-261
- Automatized C include file translation and final remarks: 262-263
- Shared libraries and dynamic loading (“Plug-Ins”): 265-271
Slide set days 4-5 (PATC)

Design Patterns (I): Creation and destruction of objects
- Polymorphic type components and polymorphic overloaded constructor, factory procedures with polymorphic dummy arguments: 3-8
- Operator overloading for polymorphic operands and function results: 9-11
- More on finalizers, the IMPURE attribute, finalization of type extensions and layered types: 12-16
- Restrictions on PURE procedures for polymorphic arguments and function results: 17

Design Patterns (II): A solution to the environment problem
- Using an interface class to solve the parameterized integration problem: 19-23
- The Strategy pattern: implementing alternative integration methods without signature changes: 24-28
- The Class Adapter pattern: smooth integration of legacy code: 29-33
- Addendum: Performance considerations for object-oriented programs: 35-38

Design Patterns (III): Error handling in complex applications
- Tracking an exception: Unwinding the stack vs. tracking along objects; the Chain of Responsibility pattern: 40-47
- Throwing and catching exceptions: 48-49
- Restoring an object’s state – the Memento pattern: 50-52

Design Patterns (IV): Maintaining a consistent view of objects
- The Observer pattern – extracting observables from a simulation: 54-57
- The Observer framework: 58-60
- Implementing the simulation: 61-65

Parallel Programming with Coarrays (I): Setting the stage
- Introduction, concept of image, resource mapping, inquiry intrinsics for image management: 67-71
- Data decomposition and local-to-global index mapping, matrix-vector multiplication as illustration: 72-75

Coarrays (II): Basics
- Need for communication, PGAS memory model, CODIMENSION attribute, coindexed accesses to coarrays (Put and Get), local accesses to coarrays: 77-81
- Synchronization requirements and rules, SYNC ALL image control statement: 82-86
- Matrix-vector example completed, analyzing the communication pattern: 87-90
- Varia: Non-trivial coindex-to-image mappings and concept of corank, supporting intrinsics, integration of type system: 91-93

TS 18508: Collective procedures
- Motivation: 95
- Simple and user-defined reduction: 96-97
- Broadcasting data to all images, general rules for collectives: 98-99
Coarrays (III): Dynamic memory
- Symmetric memory, allocatable coarrays and coarray components: 101-105
- Coarrays as TARGETs: 106
- Coarrays of container type and handling of unsymmetric memory: 107-110
- Limitations on intrinsic assignment, coarray subobjects: 111-112

Coarrays (IV): Advanced synchronization
- Image subset synchronization with the SYNC IMAGES image control statement: 114-116
- One-sided synchronization using the TS 18508 image control statements EVENT POST and EVENT WAIT, multiple posting: 117-120
- Mutual exclusion with critical regions and locks: 121-124
- Non-blocking lock and event use (EVENT_QUERY), event and lock subobjects: 125-127
- Memory fence (SYNC MEMORY) and atomic subroutines: 128-134

Coarrays (V): Considerations for type and library design
- Coarrays and object orientation – Polymorphic coarrays, execution of type- and object-bound procedures: 136-139
- Polymorphic coarray subobjects, limitations on assignment and coindexing: 140
- Coarrays as dummy arguments, corank and coindex remapping, image-dependent argument passing of coarrays: 142-143
- Limitations for coarray use in PURE and ELEMENTAL procedures: 144
- Procedure-local coarrays: 145
- Invocation of (serial) procedures with coindexed actual arguments: 146
- Coarray factory procedures: 147
- Working around limitations – Assignment overloading and type-bound communication procedures: 148-150
- Documenting synchronization behavior of library code: 151
- Interoperation of coarrays with MPI – some very basic ideas: 153-155

TS 18508: Teams of images
- Motivation: limitations of flat model for performance and program design, team concept: 157-159
- FORM TEAM statement: 160-161
- CHANGE TEAM block construct – changing execution context: 162-163
- Handling coarray data inside a CHANGE TEAM block and across the context change: 164-166
- Image selector – accessing data across team boundaries: 167
- Synchronization across teams – SYNC TEAM statement, putting things together: 168-169

Coarrays (VI): Program Termination
- The three termination variants: 171
- Error termination and the ERROR STOP statement: 172
- Normal termination via STOP and its synchronization properties, STAT argument on image control statements: 173-174
- Fail-safe execution as an optional feature in TS 18508: Failed images, the FAIL IMAGE statement, identifying failed images via STAT and the FAILED IMAGES intrinsic: 175-179
- Fail-safe execution: use of spare images in a team setup: 180