Experience from the recent HDF5 output development including OOP in FORTRAN

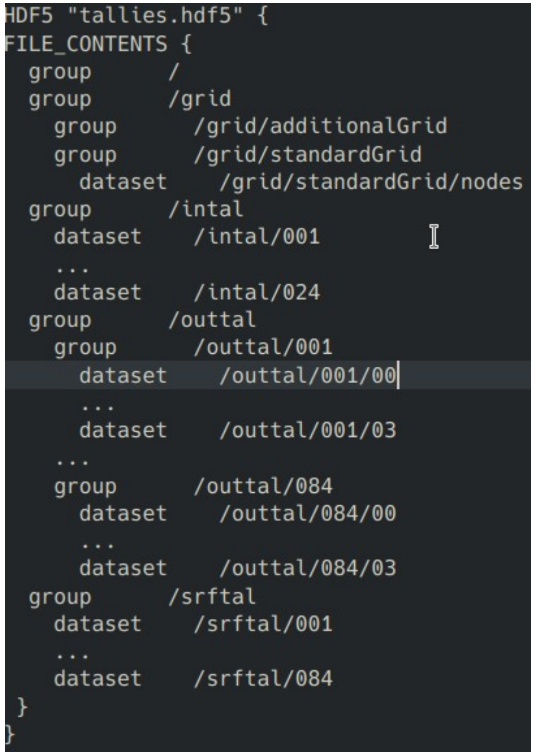
TSVV-5 VC

J. Gonzalez; 09-07-2021

HDF5 output

Structure of HDF5 output

- HDF5 is organized in datasets. A single file can contain datasets with different size and value types. Datasets can be grouped.
- Metadata can be added to the datasets to include relevant information.
- Current implementation in Eirene:
  - Branch **feature/hdf5** in Jülich repository.
  - Option in input file to activate HDF5 output.
  - Output of tallies.
  - Grid not fully implemented.
- Procedures required for HDF5 output are in module.
  - Easy to apply to other output files.



```
HDF5 "tallies.hdf5" {
FILE_CONTENTS {
 group      /
 group      /grid
   group      /grid/additionalGrid
   group      /grid/standardGrid
     dataset    /grid/standardGrid/nodes
 group      /intal
   dataset    /intal/001
   ...
   dataset    /intal/024
 group      /outtal
   group      /outtal/001
     dataset    /outtal/001/00
     ...
     dataset    /outtal/001/03
   ...
   group      /outtal/084
     dataset    /outtal/084/00
     ...
     dataset    /outtal/084/03
 group      /srftal
   dataset    /srftal/001
   ...
   dataset    /srftal/084
 }
}
```

Fig 1. Structure of HDF5 output in Eirene.

Object Oriented Programming

OOP philosophy

- In a OOP code, regardless of the language used, the basic unit to perform tasks becomes an *object*, which has its own variables and procedures.
- Objects can be extended, creating a hierarchy.
- The code is usually transparent to the object type. This means that the main program does not bother to check what type of object you are using, as it is responsibility of the object to perform tasks object dependent.

OOP simple example (I)

- Let us check how a very simple problem can be easily solved with OOP.
- An user inputs a series of geometries (circles, rectangles, triangles...) and wants to compute the area of all of them.
- First, a **TYPE** named *geometry* will be created. All other geometries will be extended from this type. The **TYPE** has a *calculateAera()* procedure which is **DEFERRED**, meaning that each extended type needs to implement it own procedure to calculate the area.

```
TYPE, PUBLIC, ABSTRACT:: geometry
  CONTAINS
    PROCEDURE(calculateArea_interface), PASS, DEFERRED:: calculateArea

END TYPE geometry

ABSTRACT INTERFACE
  FUNCTION calculateArea_interface(self) RESULT(area)
    IMPORT:: geometry
    CLASS(geometry), INTENT(in):: self
    REAL(8):: area

  END FUNCTION calculateArea_interface

END INTERFACE
```

OOP simple example (II)

- Now, each shape becomes an extension of *geometry* and has it own implementation of *calculateArea()*.
- Procedure needs to fulfill the interface defined in *geometry*.
- So, when the code wants to calculate the area of a specific geometry, it will call *calculateArea()* without knowing if the object is a rectangle or a circle.
- In a non-OOP approach, a manual way to identify geometries (usually an **INTEGER**), would have to be used and a **SELECT CASE** employed to calculate the specific area.
- This increases the code complexity and hinders its extension.

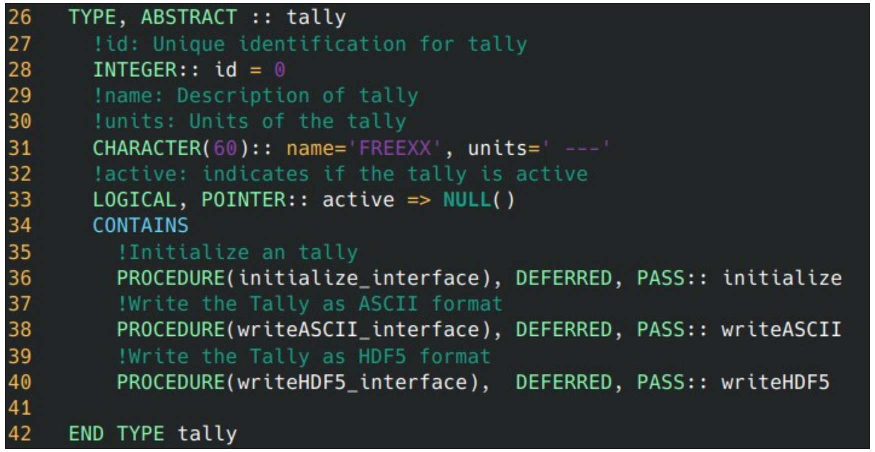
```
TYPE, PUBLIC, EXTENDS(geometry):: rectangle
  REAL(8):: l, h
  CONTAINS
    PROCEDURE, PASS:: calculateArea => calculateAreaRectangle

END TYPE rectangle

TYPE, PUBLIC, EXTENDS(geometry):: circle
  REAL(8):: r
  CONTAINS
    PROCEDURE, PASS:: calculateArea => calculateAreaCircle

END TYPE circle

FUNCTION calculateAreaRectangle(self) RESULT(area)
  IMPLICIT NONE

  CLASS(rectangle), INTENT(in):: self
  REAL(8):: area

  area = self%l * self%h

END FUNCTION calculateAreaRectangle

FUNCTION calculateAreaCircle(self) RESULT(area)
  USE constants, ONLY: PI
  IMPLICIT NONE

  CLASS(circle), INTENT(in):: self
  REAL(8):: area

  area = PI * self%r**2

END FUNCTION calculateAreaCircle
```

How basic OOP concepts are used for HDF5 and ASCII output

Simple OOP for HDF5 output in Eirene

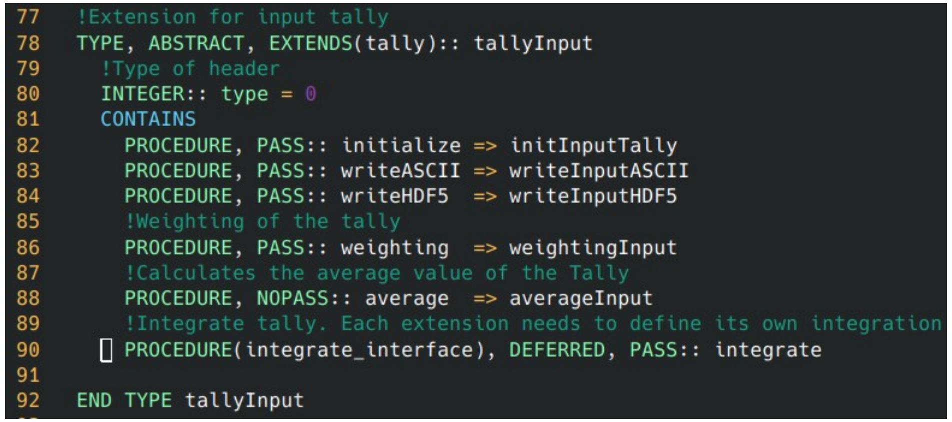
- Currently, a simple implementation of OOP is used to deal with the output of tallies in ASCII and HDF5 formats.
- Eirene has different tallies: Input, Volume Averaged (Output) and Surface Averaged (Output).
- Each tally has different units and dimension and they are written in a different way.
- New abstract type for tallies, extended for each tally type.
- Each type has information about name, units, id and pointers to the data (same structure as before regarding data management).
- Each tally type has subroutines to write its own information in ASCII or HDF5 formats.
- Reduction of **IF** and **SELECT CASE** clauses.
- Much clearer code.
- Additional improvements could be done, but require a deeper modification of Eirene.

Examples of Code



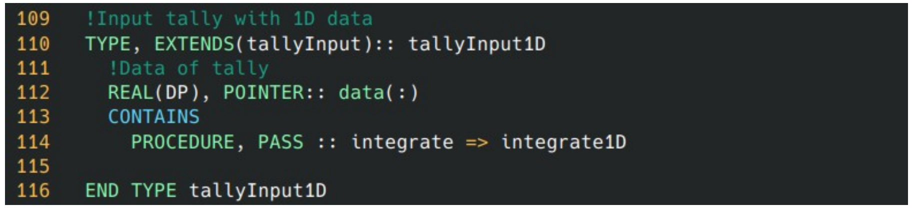
```
TYPE, ABSTRACT :: tally
  !id: Unique identification for tally
  INTEGER:: id = 0
  !name: Description of tally
  !units: Units of the tally
  CHARACTER(60):: name='FREEXX', units=' ---'
  !active: indicates if the tally is active
  LOGICAL, POINTER:: active => NULL()
  CONTAINS
    !Initialize an tally
    PROCEDURE(initialize_interface), DEFERRED, PASS:: initialize
    !Write the Tally as ASCII format
    PROCEDURE(writeASCII_interface), DEFERRED, PASS:: writeASCII
    !Write the Tally as HDF5 format
    PROCEDURE(writeHDF5_interface),  DEFERRED, PASS:: writeHDF5

END TYPE tally
```

Fig 2. Generic type for tallies.



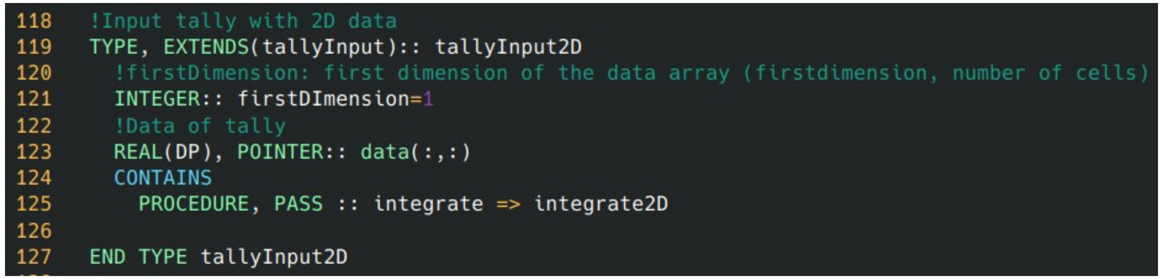
```
!Extension for input tally
TYPE, ABSTRACT, EXTENDS(tally):: tallyInput
  !Type of header
  INTEGER:: type = 0
  CONTAINS
    PROCEDURE, PASS:: initialize => initInputTally
    PROCEDURE, PASS:: writeASCII => writeInputASCII
    PROCEDURE, PASS:: writeHDF5  => writeInputHDF5
    !Weighting of the tally
    PROCEDURE, PASS:: weighting  => weightingInput
    !Calculates the average value of the Tally
    PROCEDURE, NOPASS:: average  => averageInput
    !Integrate tally. Each extension needs to define its own integration
    PROCEDURE(integrate_interface), DEFERRED, PASS:: integrate

END TYPE tallyInput
```

Fig 3. Extension for input tallies.



```
!Input tally with 1D data
TYPE, EXTENDS(tallyInput):: tallyInput1D
  !Data of tally
  REAL(DP), POINTER:: data(:)
  CONTAINS
    PROCEDURE, PASS :: integrate => integrate1D

END TYPE tallyInput1D
```

Fig 4. Input tally for 1D data.



```
!Input tally with 2D data
TYPE, EXTENDS(tallyInput):: tallyInput2D
  !firstDimension: first dimension of the data array (firstdimension, number of cells)
  INTEGER:: firstDImension=1
  !Data of tally
  REAL(DP), POINTER:: data(:,:)
  CONTAINS
    PROCEDURE, PASS :: integrate => integrate2D

END TYPE tallyInput2D
```

Fig 5. Input tally for 2D data.

Thank you for your attention

J. Gonzalez | TSVV-5 VC

DIFFER