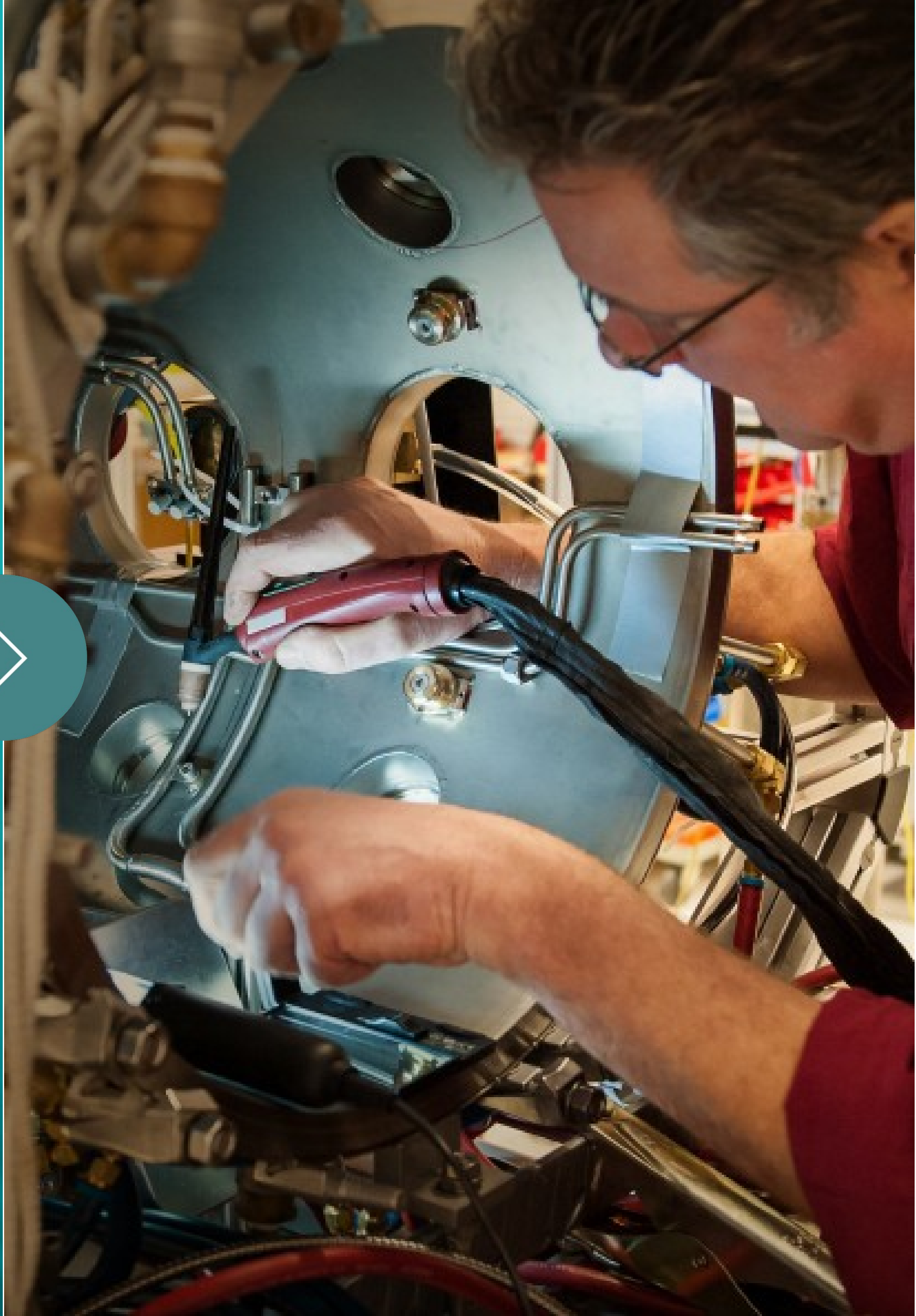
Variable Grouping

Eirene Streamlining Code Camp

J. Gonzalez; 04-11-2021



Objective

Objective of the session

1)Explain briefly how grouping variables could help Eirene development.

2)Present a recent development of Eirene in which abstract TYPEs were used.

3)Propose an exercise to get everyone familiar with TYPE.

4)Studying possibilities of grouping variables in Eirene within modules (internal variables).

Why use TYPEs to organize variables

- **Clearer code**: when calling a procedure with multiple related variables, only the main variable will be passed.
- It helps to **identify variables** when reading and debugging code. Giving a structure to variables and “linking” them provides a helpful way to identify them.
- Better way to store and access arrays. For example:

```
X(1:nodes), Y(1:nodes), Z(1:nodes) => node(1:nodes)%X, node(1:nodes)%Y, node(1:nodes)%Z
```

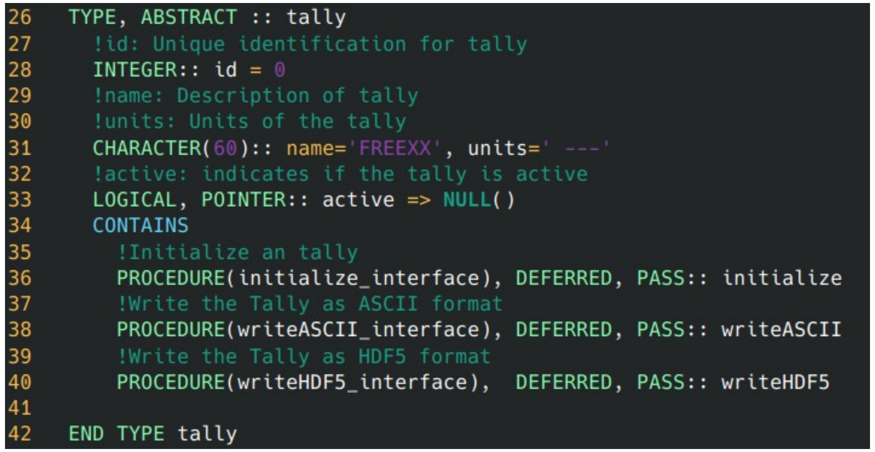
- Code is **easier to modify**, specially with an OOP philosophy in mind. It is easier to expand a TYPE with a related variable than to add a new variable, modify interfaces, calls, modules…
- It could provide guidance to manage **Eirene input**. The way the input variables are structured could be rewritten in TYPEs, helping with default values and also having an equivalence with new JSON format.

Previous experiences: Tallies for ASCII and HDF5 outputs

Simple OOP for HDF5 output in Eirene

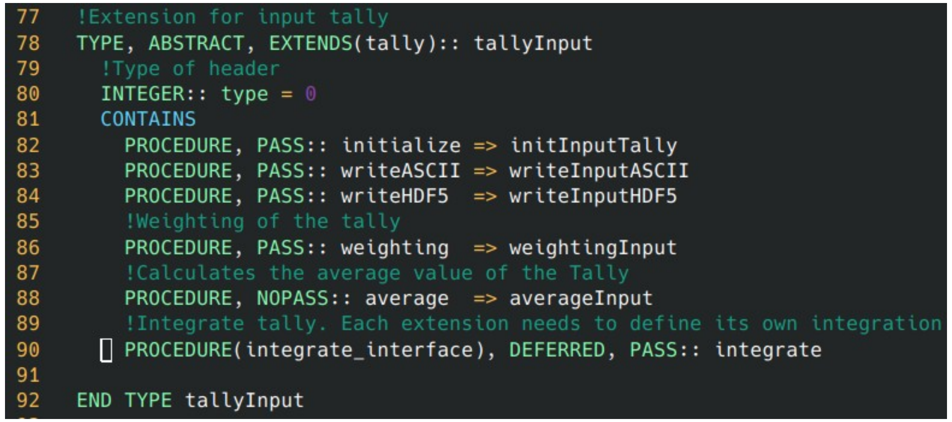
- Currently, a simple implementation of OOP is used to deal with the output of tallies in ASCII and HDF5 formats.
- Eirene has different tallies: Input, Volume Averaged (Output) and Surface Averaged (Output).
- Each tally has different units and dimension and they are written in a different way.
- New 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



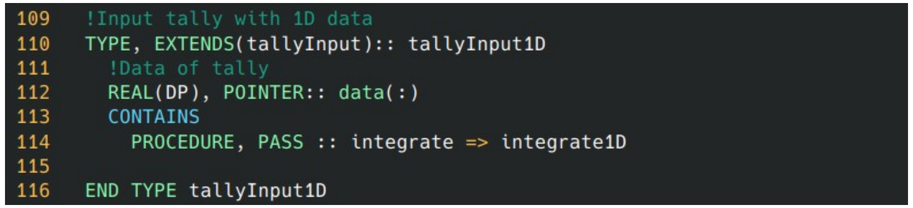
```fortran
TYPE, ABSTRACT :: tally
  !id: Unique identification for tally
  INTEGER:: id = 0
  !name: Description of tally
  !units: Units of the tally
  CHARACTER(60):: name='FREEXX', units=' ---'
  !active: indicates if the tally is active
  LOGICAL, POINTER:: active => NULL()
  CONTAINS
    !Initialize 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.



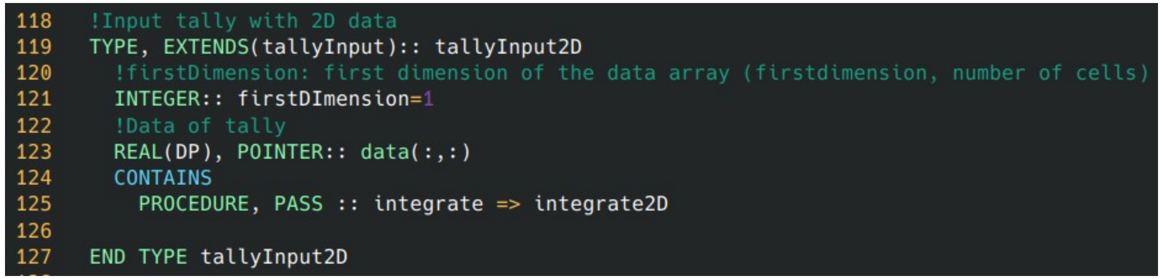
```fortran
!Extension for input tally
TYPE, ABSTRACT, EXTENDS(tally):: tallyInput
  !Type of header
  INTEGER:: type = 0
  CONTAINS
    PROCEDURE, PASS:: initialize => initInputTally
    PROCEDURE, PASS:: writeASCII => writeInputASCII
    PROCEDURE, PASS:: writeHDF5  => writeInputHDF5
    !Weighting 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.



```fortran
!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.



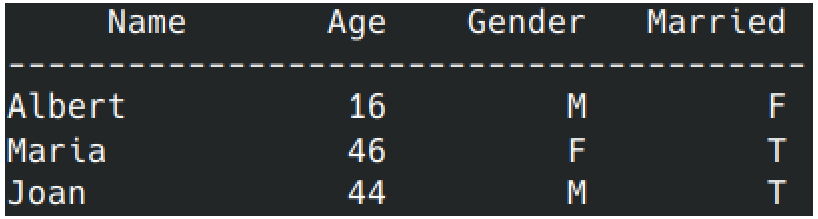
```fortran
!Input tally with 2D data
TYPE, EXTENDS(tallyInput):: tallyInput2D
  !firstDimension: first dimension of the data array (firstdimension, number of cells)
  INTEGER:: firstDImension=1
  !Data of tally
  REAL(DP), POINTER:: data(:,:)
  CONTAINS
    PROCEDURE, PASS :: integrate => integrate2D

END TYPE tallyInput2D
```

Fig 5. Input tally for 2D data.

A guided example

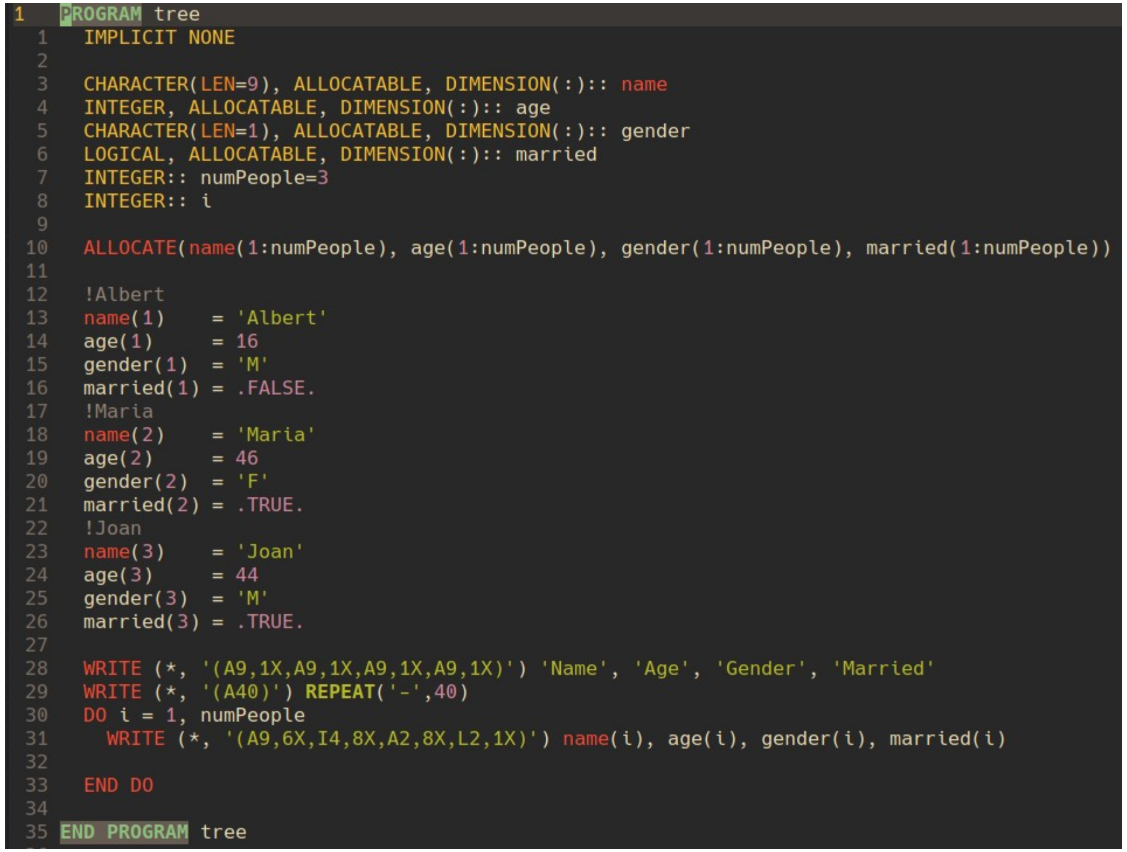
A ‘simple’ example: A Genealogical Tree (v0)

- *WARNING*: Fortran is not the best code for this type of example, but enough to illustrate the concepts.
- We want to print information about persons and their relations.
- One option: Create an array for each variable.



```
     Name        Age     Gender    Married
----------------------------------------
Albert            16          M          F
Maria             46          F          T
Joan              44          M          T
```

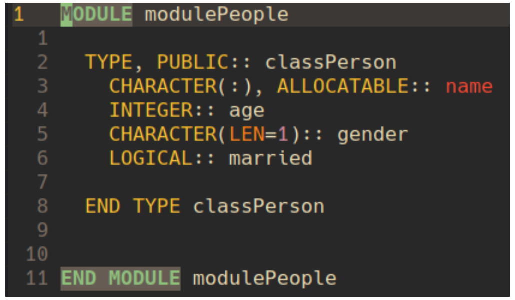
Bulky, difficult to expand.
No relation between variables.
Multiple access to different arrays.
All *name* have same length.



```fortran
PROGRAM tree
  IMPLICIT NONE

  CHARACTER(LEN=9), ALLOCATABLE, DIMENSION(:):: name
  INTEGER, ALLOCATABLE, DIMENSION(:):: age
  CHARACTER(LEN=1), ALLOCATABLE, DIMENSION(:):: gender
  LOGICAL, ALLOCATABLE, DIMENSION(:):: married
  INTEGER:: numPeople=3
  INTEGER:: i

  ALLOCATE(name(1:numPeople), age(1:numPeople), gender(1:numPeople), married(1:numPeople))

  !Albert
  name(1)    = 'Albert'
  age(1)     = 16
  gender(1)  = 'M'
  married(1) = .FALSE.
  !Maria
  name(2)    = 'Maria'
  age(2)     = 46
  gender(2)  = 'F'
  married(2) = .TRUE.
  !Joan
  name(3)    = 'Joan'
  age(3)     = 44
  gender(3)  = 'M'
  married(3) = .TRUE.

  WRITE (*, '(A9,1X,A9,1X,A9,1X,A9,1X)') 'Name', 'Age', 'Gender', 'Married'
  WRITE (*, '(A40)') REPEAT('-',40)
  DO i = 1, numPeople
    WRITE (*, '(A9,6X,I4,8X,A2,8X,L2,1X)') name(i), age(i), gender(i), married(i)

  END DO

END PROGRAM tree
```

A ‘simple’ example: A Genealogical Tree (v1)

- First improvement: group related variables in a new TYPE.

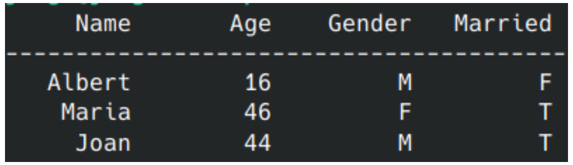


```fortran
MODULE modulePeople

  TYPE, PUBLIC:: classPerson
    CHARACTER(:), ALLOCATABLE:: name
    INTEGER:: age
    CHARACTER(LEN=1):: gender
    LOGICAL:: married

  END TYPE classPerson


END MODULE modulePeople
```

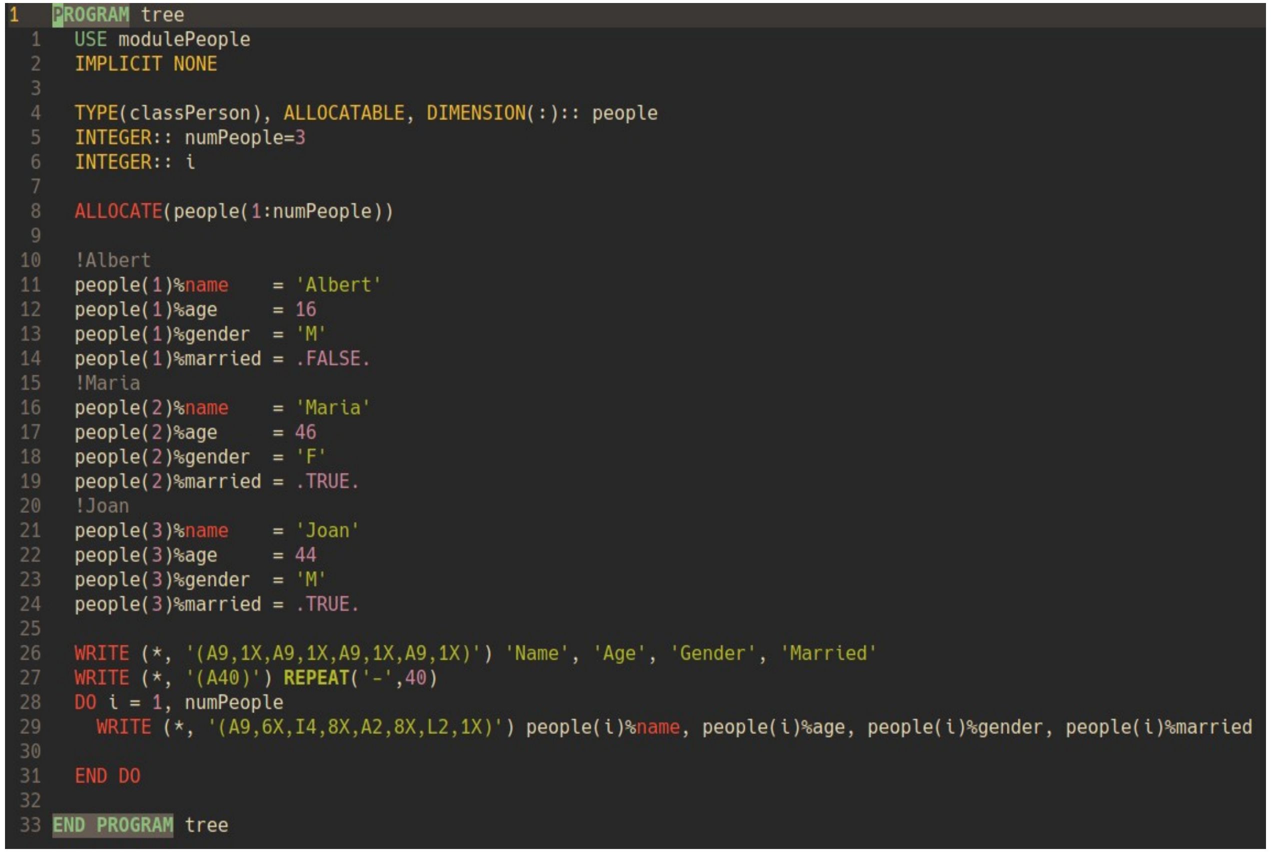
Little bit clearer.

Related variables are grouped.

Name is a variable length.



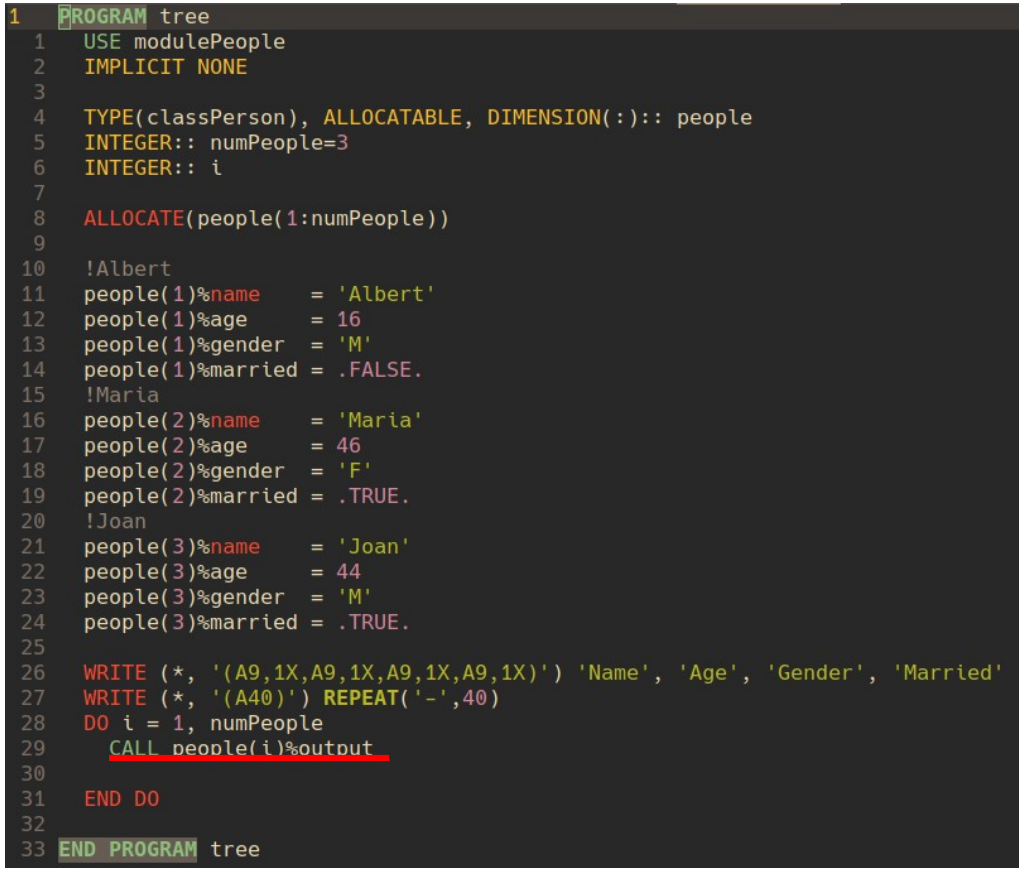
```
     Name       Age    Gender   Married
----------------------------------------
   Albert        16         M         F
    Maria        46         F         T
     Joan        44         M         T
```



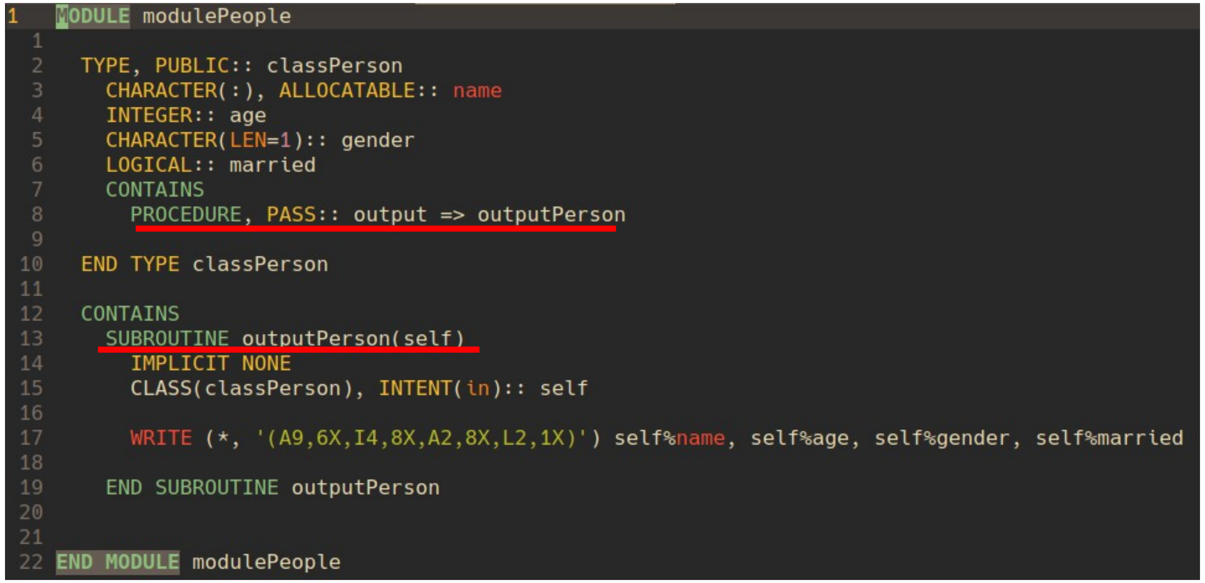
```fortran
PROGRAM tree
  USE modulePeople
  IMPLICIT NONE

  TYPE(classPerson), ALLOCATABLE, DIMENSION(:):: people
  INTEGER:: numPeople=3
  INTEGER:: i

  ALLOCATE(people(1:numPeople))

  !Albert
  people(1)%name    = 'Albert'
  people(1)%age     = 16
  people(1)%gender  = 'M'
  people(1)%married = .FALSE.
  !Maria
  people(2)%name    = 'Maria'
  people(2)%age     = 46
  people(2)%gender  = 'F'
  people(2)%married = .TRUE.
  !Joan
  people(3)%name    = 'Joan'
  people(3)%age     = 44
  people(3)%gender  = 'M'
  people(3)%married = .TRUE.

  WRITE (*, '(A9,1X,A9,1X,A9,1X,A9,1X)') 'Name', 'Age', 'Gender', 'Married'
  WRITE (*, '(A40)') REPEAT('-',40)
  DO i = 1, numPeople
    WRITE (*, '(A9,6X,I4,8X,A2,8X,L2,1X)') people(i)%name, people(i)%age, people(i)%gender, people(i)%married

  END DO

END PROGRAM tree
```

A ‘simple’ example: A Genealogical Tree (v1.1)

- A little improvement, offload printing to the module:



```
PROGRAM tree
  USE modulePeople
  IMPLICIT NONE

  TYPE(classPerson), ALLOCATABLE, DIMENSION(:):: people
  INTEGER:: numPeople=3
  INTEGER:: i

  ALLOCATE(people(1:numPeople))

  !Albert
  people(1)%name    = 'Albert'
  people(1)%age     = 16
  people(1)%gender  = 'M'
  people(1)%married = .FALSE.
  !Maria
  people(2)%name    = 'Maria'
  people(2)%age     = 46
  people(2)%gender  = 'F'
  people(2)%married = .TRUE.
  !Joan
  people(3)%name    = 'Joan'
  people(3)%age     = 44
  people(3)%gender  = 'M'
  people(3)%married = .TRUE.

  WRITE (*, '(A9,1X,A9,1X,A9,1X,A9,1X)') 'Name', 'Age', 'Gender', 'Married'
  WRITE (*, '(A40)') REPEAT('-',40)
  DO i = 1, numPeople
    CALL people(i)%output

  END DO

END PROGRAM tree
```



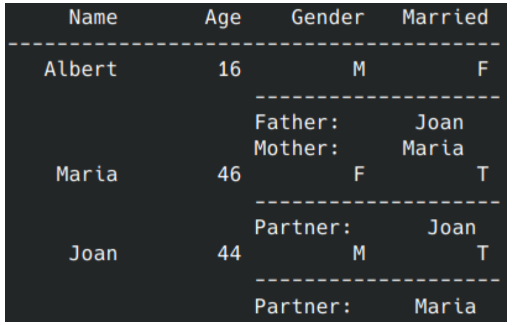
```
MODULE modulePeople

  TYPE, PUBLIC:: classPerson
    CHARACTER(:), ALLOCATABLE:: name
    INTEGER:: age
    CHARACTER(LEN=1):: gender
    LOGICAL:: married
    CONTAINS
      PROCEDURE, PASS:: output => outputPerson

  END TYPE classPerson

  CONTAINS
    SUBROUTINE outputPerson(self)
      IMPLICIT NONE
      CLASS(classPerson), INTENT(in):: self

      WRITE (*, '(A9,6X,I4,8X,A2,8X,L2,1X)') self%name, self%age, self%gender, self%married

    END SUBROUTINE outputPerson


END MODULE modulePeople
```

The main code does not have to worry about the elements to print as it is responsibility of the module.

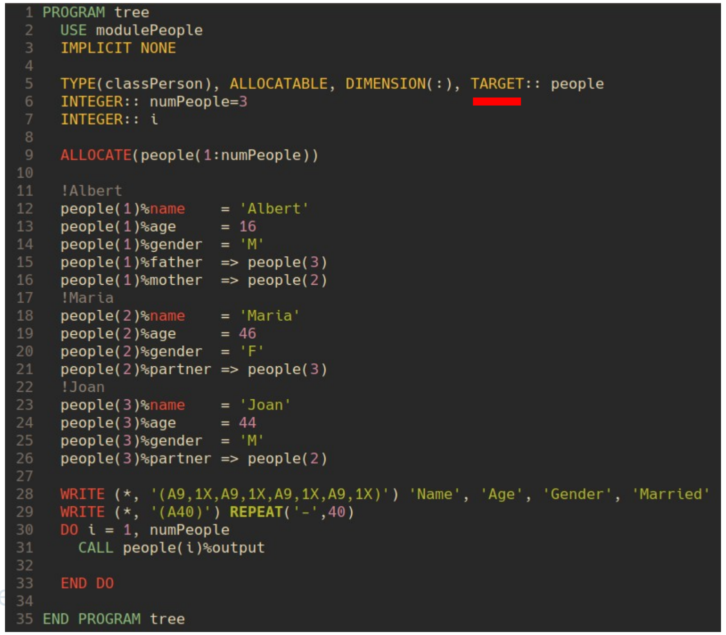
A ‘simple’ example: A Genealogical Tree (v2)

- Okay, but what about relations?

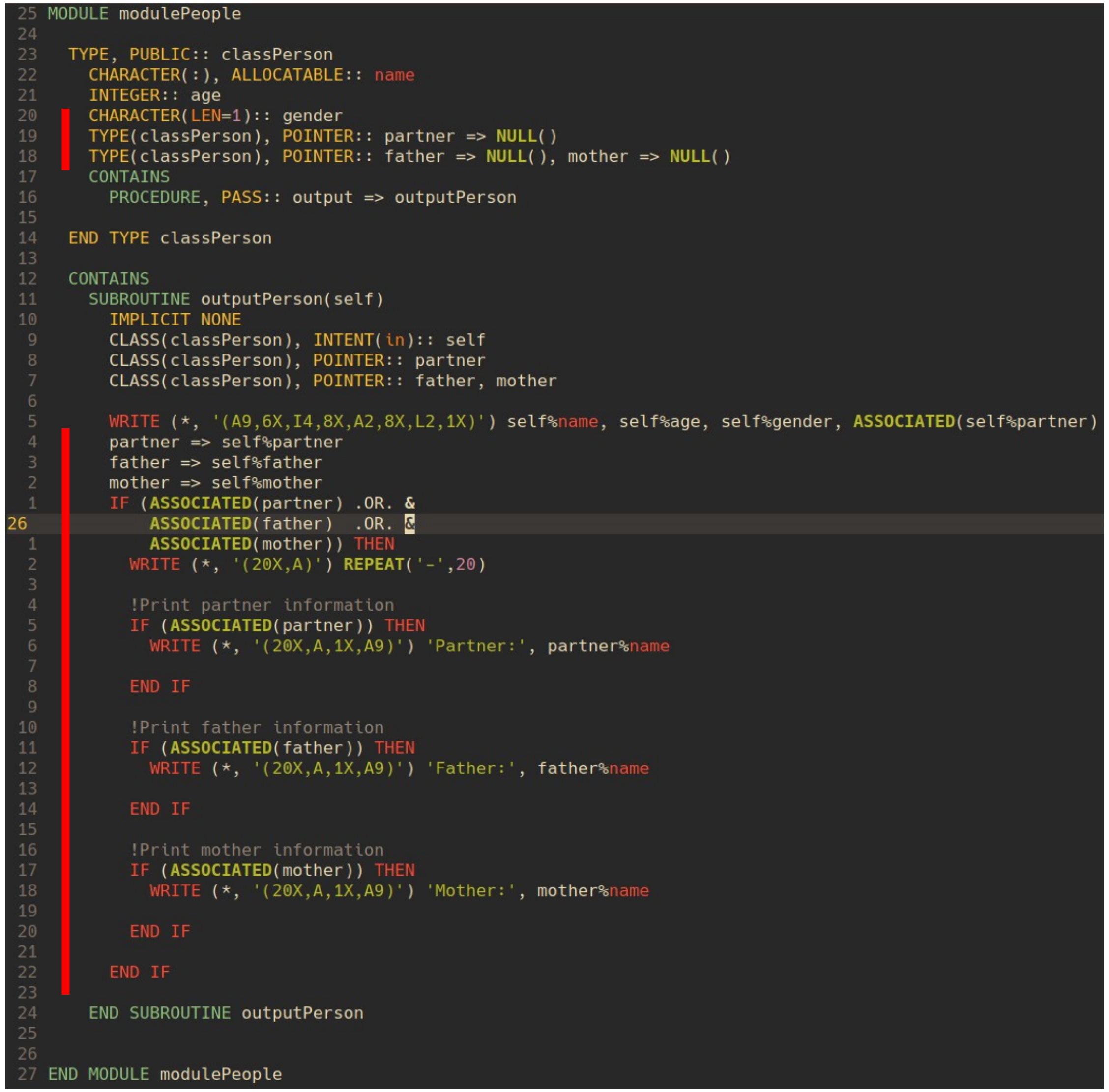
Removed the *married* logical.
Complex printing procedure is encapsulated.
Minimum changes to main code.



```
    Name         Age    Gender   Married
 ----------------------------------------
  Albert          16         M         F
                    --------------------
                    Father:        Joan
                    Mother:       Maria
   Maria          46         F         T
                    --------------------
                    Partner:       Joan
    Joan          44         M         T
                    --------------------
                    Partner:      Maria
```



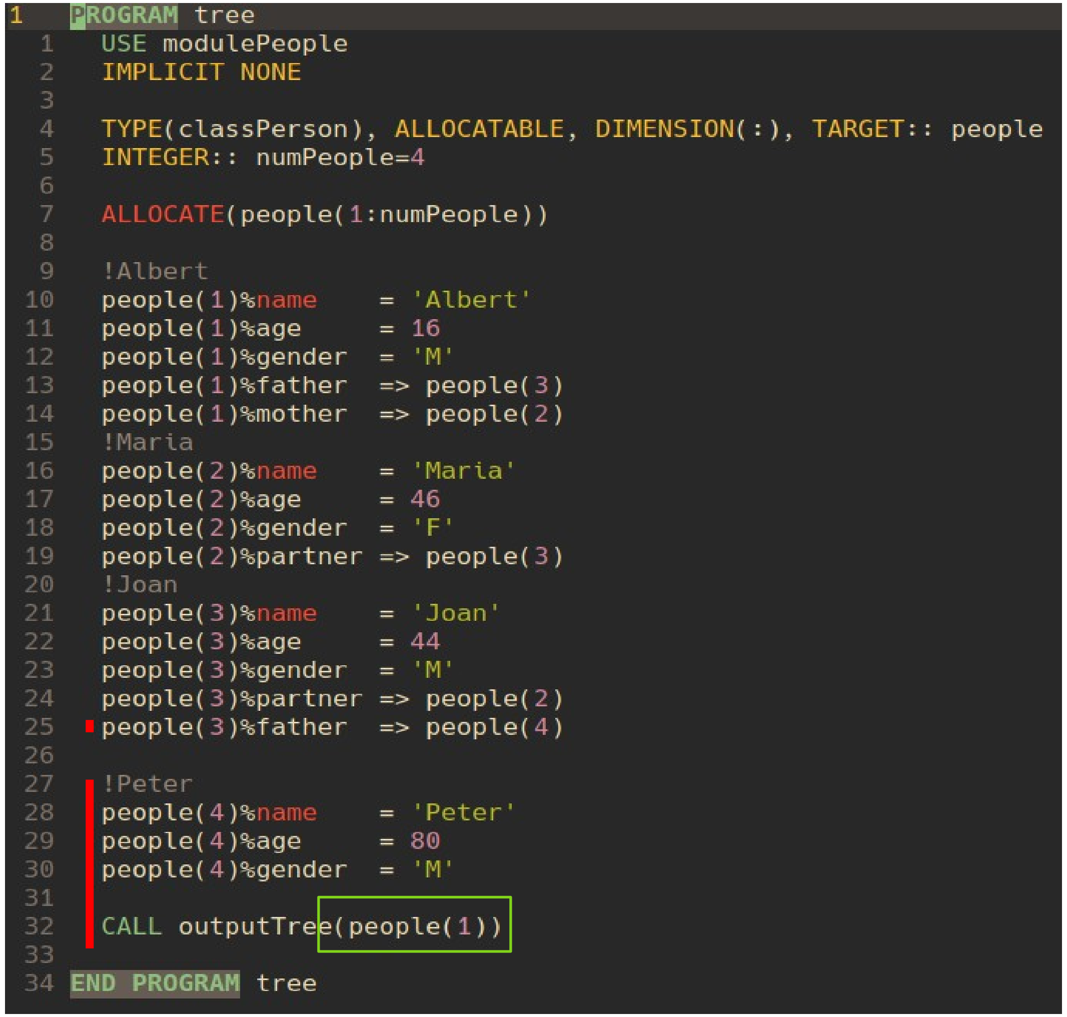
```
PROGRAM tree
  USE modulePeople
  IMPLICIT NONE

  TYPE(classPerson), ALLOCATABLE, DIMENSION(:), TARGET:: people
  INTEGER:: numPeople=3
  INTEGER:: i

  ALLOCATE(people(1:numPeople))

  !Albert
  people(1)%name    = 'Albert'
  people(1)%age     = 16
  people(1)%gender  = 'M'
  people(1)%father  => people(3)
  people(1)%mother  => people(2)
  !Maria
  people(2)%name    = 'Maria'
  people(2)%age     = 46
  people(2)%gender  = 'F'
  people(2)%partner => people(3)
  !Joan
  people(3)%name    = 'Joan'
  people(3)%age     = 44
  people(3)%gender  = 'M'
  people(3)%partner => people(2)

  WRITE (*, '(A9,1X,A9,1X,A9,1X,A9,1X)') 'Name', 'Age', 'Gender', 'Married'
  WRITE (*, '(A40)') REPEAT('-',40)
  DO i = 1, numPeople
    CALL people(i)%output

  END DO

END PROGRAM tree
```



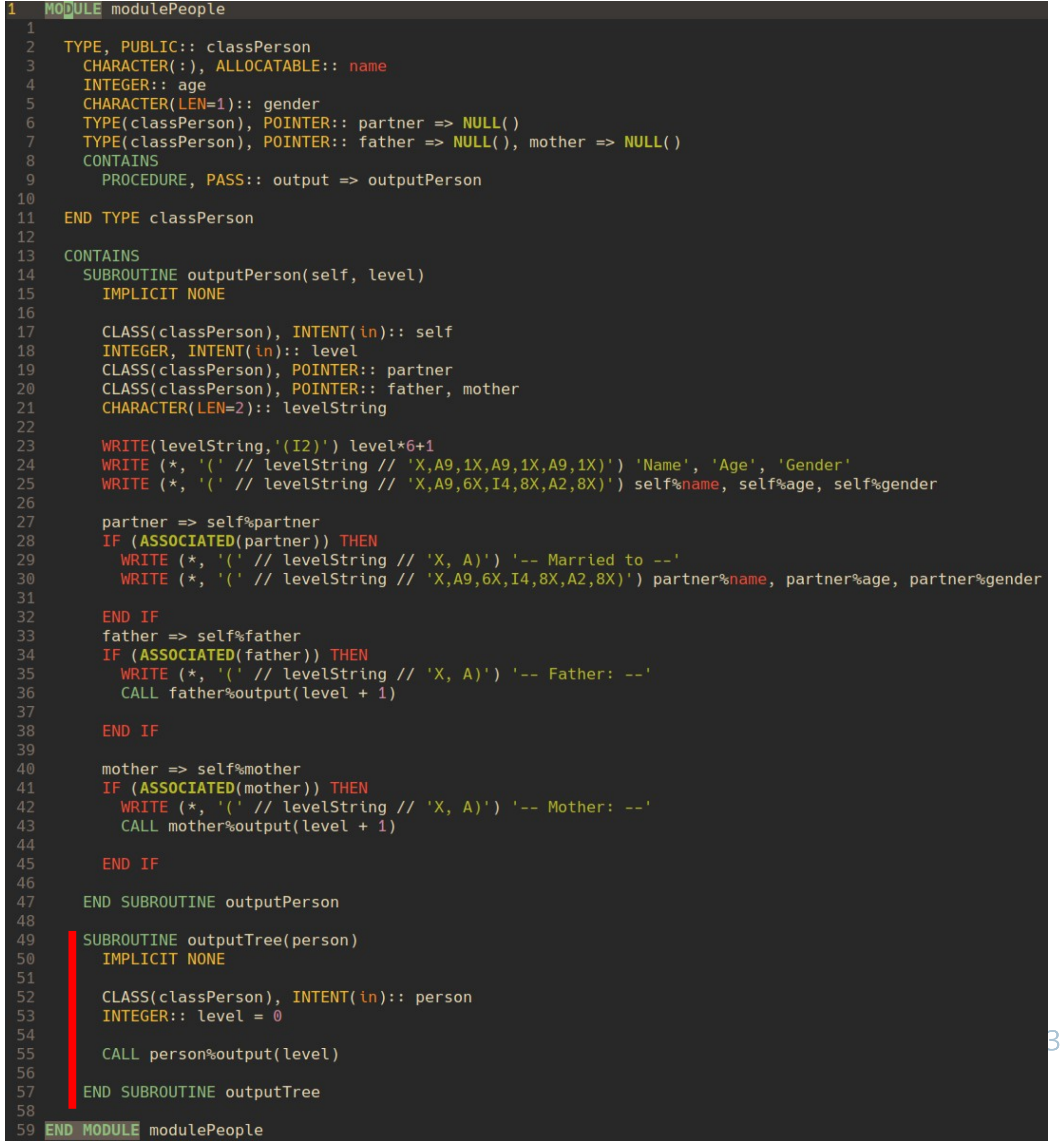
```
MODULE modulePeople

  TYPE, PUBLIC:: classPerson
    CHARACTER(:), ALLOCATABLE:: name
    INTEGER:: age
    CHARACTER(LEN=1):: gender
    TYPE(classPerson), POINTER:: partner => NULL()
    TYPE(classPerson), POINTER:: father => NULL(), mother => NULL()
    CONTAINS
      PROCEDURE, PASS:: output => outputPerson

  END TYPE classPerson

  CONTAINS
    SUBROUTINE outputPerson(self)
      IMPLICIT NONE
      CLASS(classPerson), INTENT(in):: self
      CLASS(classPerson), POINTER:: partner
      CLASS(classPerson), POINTER:: father, mother

      WRITE (*, '(A9,6X,I4,8X,A2,8X,L2,1X)') self%name, self%age, self%gender, ASSOCIATED(self%partner)
      partner => self%partner
      father => self%father
      mother => self%mother
      IF (ASSOCIATED(partner) .OR. &
          ASSOCIATED(father)  .OR. &
          ASSOCIATED(mother)) THEN
        WRITE (*, '(20X,A)') REPEAT('-',20)

        !Print partner information
        IF (ASSOCIATED(partner)) THEN
          WRITE (*, '(20X,A,1X,A9)') 'Partner:', partner%name

        END IF

        !Print father information
        IF (ASSOCIATED(father)) THEN
          WRITE (*, '(20X,A,1X,A9)') 'Father:', father%name

        END IF

        !Print mother information
        IF (ASSOCIATED(mother)) THEN
          WRITE (*, '(20X,A,1X,A9)') 'Mother:', mother%name

        END IF

      END IF

    END SUBROUTINE outputPerson


END MODULE modulePeople
```

A ‘simple’ example: A Genealogical Tree (v3)

- Now, let us have a ‘tree’.



```fortran
PROGRAM tree
  USE modulePeople
  IMPLICIT NONE

  TYPE(classPerson), ALLOCATABLE, DIMENSION(:), TARGET:: people
  INTEGER:: numPeople=4

  ALLOCATE(people(1:numPeople))

  !Albert
  people(1)%name    = 'Albert'
  people(1)%age     = 16
  people(1)%gender  = 'M'
  people(1)%father  => people(3)
  people(1)%mother  => people(2)
  !Maria
  people(2)%name    = 'Maria'
  people(2)%age     = 46
  people(2)%gender  = 'F'
  people(2)%partner => people(3)
  !Joan
  people(3)%name    = 'Joan'
  people(3)%age     = 44
  people(3)%gender  = 'M'
  people(3)%partner => people(2)
  people(3)%father  => people(4)

  !Peter
  people(4)%name    = 'Peter'
  people(4)%age     = 80
  people(4)%gender  = 'M'

  CALL outputTree(people(1))

END PROGRAM tree
```

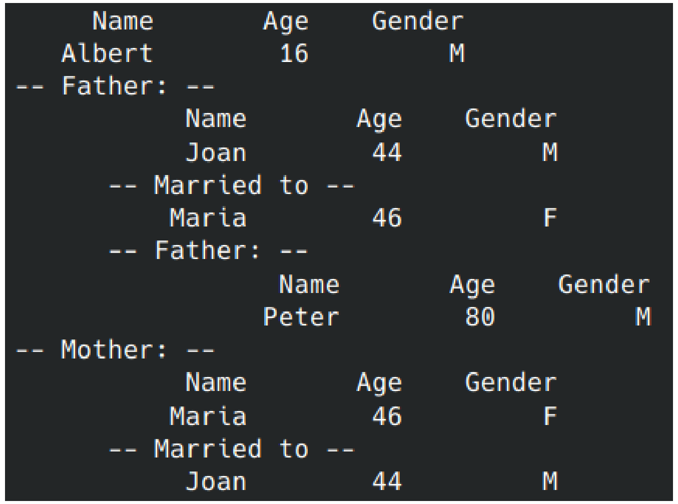


```fortran
MODULE modulePeople

  TYPE, PUBLIC:: classPerson
    CHARACTER(:), ALLOCATABLE:: name
    INTEGER:: age
    CHARACTER(LEN=1):: gender
    TYPE(classPerson), POINTER:: partner => NULL()
    TYPE(classPerson), POINTER:: father => NULL(), mother => NULL()
    CONTAINS
      PROCEDURE, PASS:: output => outputPerson

  END TYPE classPerson

  CONTAINS
    SUBROUTINE outputPerson(self, level)
      IMPLICIT NONE

      CLASS(classPerson), INTENT(in):: self
      INTEGER, INTENT(in):: level
      CLASS(classPerson), POINTER:: partner
      CLASS(classPerson), POINTER:: father, mother
      CHARACTER(LEN=2):: levelString

      WRITE(levelString,'(I2)') level*6+1
      WRITE (*, '(' // levelString // 'X,A9,1X,A9,1X,A9,1X)') 'Name', 'Age', 'Gender'
      WRITE (*, '(' // levelString // 'X,A9,6X,I4,8X,A2,8X)') self%name, self%age, self%gender

      partner => self%partner
      IF (ASSOCIATED(partner)) THEN
        WRITE (*, '(' // levelString // 'X, A)') '-- Married to --'
        WRITE (*, '(' // levelString // 'X,A9,6X,I4,8X,A2,8X)') partner%name, partner%age, partner%gender

      END IF
      father => self%father
      IF (ASSOCIATED(father)) THEN
        WRITE (*, '(' // levelString // 'X, A)') '-- Father: --'
        CALL father%output(level + 1)

      END IF

      mother => self%mother
      IF (ASSOCIATED(mother)) THEN
        WRITE (*, '(' // levelString // 'X, A)') '-- Mother: --'
        CALL mother%output(level + 1)

      END IF

    END SUBROUTINE outputPerson

    SUBROUTINE outputTree(person)
      IMPLICIT NONE

      CLASS(classPerson), INTENT(in):: person
      INTEGER:: level = 0

      CALL person%output(level)

    END SUBROUTINE outputTree

END MODULE modulePeople
```

A ‘simple’ example: A Genealogical Tree (v3)

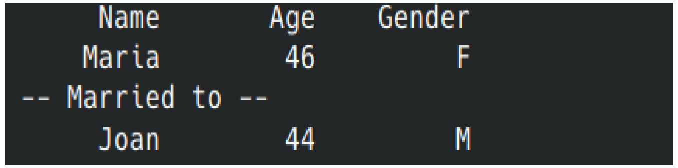
- Different trees for different *people(i)*

people(1)



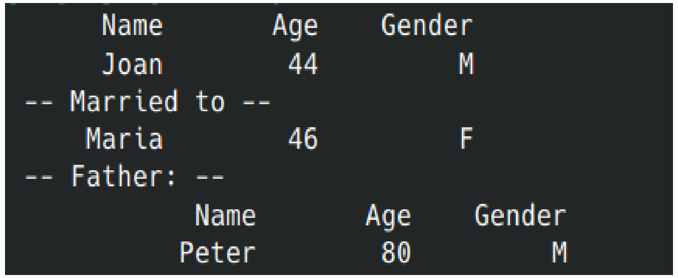
```
   Name        Age     Gender
  Albert        16          M
-- Father: --
           Name        Age     Gender
           Joan         44          M
      -- Married to --
          Maria         46          F
      -- Father: --
                 Name        Age     Gender
                Peter         80          M
-- Mother: --
           Name        Age     Gender
          Maria         46          F
      -- Married to --
           Joan         44          M
```

people(2)



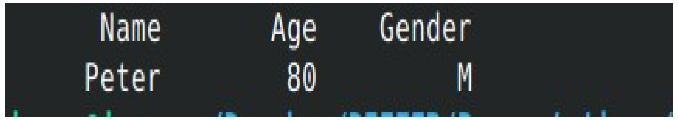
```
   Name        Age     Gender
  Maria         46          F
-- Married to --
   Joan         44          M
```

people(3)



```
   Name        Age     Gender
   Joan         44          M
-- Married to --
  Maria         46          F
-- Father: --
         Name        Age     Gender
        Peter         80          M
```

people(4)



```
   Name        Age     Gender
  Peter         80          M
```

We don’t know how many sublevels we will have to plot for each person, we just **request** the module to print it and he takes care of everything.
If we wanted to add information, only the module will be modified.

Opportunities in Eirene

Collisions

- Similar structure: cross-section, species involved, energy lost…
- It could help to organize input file.
- Multiple collision types, so maybe extensions of types are required.

Particles

- Test particles have a large number of parameters to be traced: position (3D), velocity (3D), cell in which they are located, weight...

General positions and velocities

- Usually, positions and velocities in Eirene are referred to with X, Y, Z (or VX, VY, VZ), usually deriving in large arrays.
- These could be grouped easily in types.

Geometry (maybe IMAS related)

- Geometry is a good candidate for variable grouping as Finite Elements are normally treated as a hierarchy.
- However, it will be good to have this development in line with GGD.

Thank you for your attention

J. Gonzalez | Eirene Code Camp 2021

DIFFER

Title

Title

- Text