Exercises for Object-Oriented Programming in Fortran

Arjen Markus

June 30, 2020

This document belongs to the FortranCon 2020 workshop on *Object-Oriented Programming in Fortran*. It describes a set of exercises to get acquainted with this style of programming.

The exercises can be done with any compiler that conforms to the Fortran 2003 standard. For efficiency several source files are available that define various modules and classes and can be used either as "parent classes" or as templates.

Exercise 1: Moving average

Create a class that *incrementally* calculates a moving average from a timeseries.

The class is meant to take data one by one and return the moving average of these data. MOre specifically:

- An object of this class is initialised with the number of samples that will make up the moving average – the "window".
- It takes new data one at a time and calculates the moving average from the previous data and the new input (the method is called `put_value`).
- Another method, `get_average`, returns the moving average at that moment.

Create a new class by extension that can also report the minimum, maximum and standard deviation (for the data within the window).

Exercise 2: Thermostat

Create a small program to simulate a room with a basic thermostat.

Here are the specifications:

- The room can be modelled as a well-mixed vessel where heat is exchanged with the outside world according to the difference in temperature.

- To keep the temperature as close to the set temperature as possible, the thermostat can turn on or off a heating device that has only a single output value.
- The set temperature is a function of the day (a daily cycle).
- The outside temperature is a function of time (variation within the day and over the days of the year).

The temperature in the room can be described via the following differential equation:

$$\frac{dT}{dt} = k(T_{out} - T) + H(t)/\rho c_p V \tag{1}$$

where:

k	heat exchange coefficient
T	temperature in the room
T_{out}	outside temperature
$H(t)$	output from the heater
$\rho c_p V$	scale factor, actually the heat capacity of the room

To keep the program flexible and easily extensible, use (dedicated) objects for:

- the outside temperature – it could be read from a file or calculated from a formula.
- the thermostat which controls the heater – the set temperature varies over the day but a smarter thermostat might want to keep track of the amount of heat or even be smart enough to start or stop the heater before the set temperature changes.

Do not worry abot the numerical values: we are not looking for a realistic simulation.

Exercise 3: Replicating objects

Create a program where objects show dynamic behaviour, including ”dying” and ”reproducing”.

The idea is simple:
The objects have a certain ”survival” expectancy, each time step there is a chance that they disappear. But if they survive long enough, they split into two brandnew objects. These new objects have an ”age” zero and so really start anew.

Notes:

- You may decide for yourself whether to use an array of such objects (simplest solution: just decide before how many objects can be ”alive” at any one time and stop if there appear more) or to use a linked list (see `linkedlist.f90` for an example implementation).

- The survival rate and the age at which they can reproduce determine whether the population will expand or decline. Select them with some care – or experiment.
- The time and the age may simply be integers – you would count the number of time steps.
- Start the calculation with a small but decent number of objects.
- Be sure to clean up the memory when an object ”dies”.

Exercise 4: Universal container

Create a container class that can store any type of data.

The rather strict rules in Fortran regarding types sometimes seem to hinder rather than help. However, *unlimited polymorphic variables* can be used to get around that. Use them to store any type of data in an array and supply methods to retrieve them by index.

Here is a code snippet to illustrate the sort of use you could make of this:

```
integer                       :: participants
character(len=:), allocatable :: name
logical                       :: error
...
call storage%store( 1, 123 )
call storage%store( 2, "Zuerich" )
...
!
! Retrieve the data
!
call storage%retrieve( 1, participants, error )
if ( error ) then
    write(*,*) 'Element 1 is NOT an integer value!'
endif

call storage%retrieve( 1, name, error )
if ( error ) then
    write(*,*) 'Element 2 is NOT a string!'
endif
...
```

The code snippet illustrates that the container should provide a check on the correct type of the stored information.

Exercise 5: Abstract framework

Create an abstract class for solving systems of differential equations.

The set of equations to solve is:

$$\frac{d}{dt}\underline{x} = \underline{f}(t, \underline{x}) \tag{2}$$

where $\underline{x}$ is the vector of dependent variables and $\underline{f}$ is a vector-valued function.

A simple example could be (a dampened oscillation):

$$\begin{aligned} \frac{dx}{dt} &= y && (3) \\ \frac{dy}{dt} &= -kx - ry && (4) \end{aligned}$$

The framework allows the user to:

- Specify the righthand side – the function $\underline{f}$.
- Define a print interval and the output file to write to.
- Solve the equations over a given interval. The input values for $\underline{x}$ constitute the initial condition. The output values constitute the result at the end of the interval.

Specific classes should implement such numerical methods as that of Euler or Heun.

Implementation note: the number of equations is equal to the number of elements in the `x`-vector passed to the `solve` method, but there is no way to formally require that the function that was specified returns a vector of derivatives with the same number of elements.