Transactions of the Korean Nuclear Society Spring Meeting
Taebaek, Korea, May 26-27, 2011

Implementation of Neutronics Analysis Code using the Features of Object Oriented Programming via Fortran90/95

Tae Young Han, Beom Jin Cho
KEPCO Nuclear Fuel, 1047 Daedukdaero, Yuseong-gu, Daejeon, Korea, 305-353
**Corresponding author: tyhan@knfc.co.kr*

1. Introduction

The object-oriented programming (OOP) [1] concept was radically established after 1990s and successfully involved in Fortran 90/95 [2]. The features of OOP are such as the information hiding, encapsulation, modularity and inheritance, which lead to producing code that satisfy three R's: reusability, reliability and readability. The major OOP concepts, however, except *Module* are not mainly used in neutronics analysis codes even though the code was written by Fortran 90/95.

In this work, we show that the OOP concept can be employed to develop the neutronics analysis code, ASTRA1D (Advanced Static and Transient Reactor Analyzer for 1-Dimension), via Fortran90/95 and those can be more efficient and reasonable programming methods.

2. Class Design

The most important thing in the object oriented programming is the abstraction of data and the definition of class. The abstraction means that problems or models are simplified to variables and their functional methods. Also, the class stands for self-sufficient modules involving variables and methods obtained through the process of the data abstraction. Since the concept of OOP in Fortran 90/95, however, is basically imperfect, such notations can be incompletely but sufficiently implemented by user-defined data type and *Module*.

In next section, it is presented how to construct the classes from the neutronics analysis model and how to compose variables and methods for the classes including the features of OOP.

2.1 Class Hierarchy

Node and *Face* class can be constructed as the elementary user-defined data type in diffusion equation nodal solver. Here, *Node* means a neutronics mesh in the solver and *Face* means a boundary between two neighboring nodes. Then, *FuelAssembly* class can be composed of *Node* class and *Face* class and it obviously means fuel assembly in reactor core. Lastly, *Reactor* class designed as a reactor core is composed of *FuelAssembly* class, *ControlAssembly* class and *CoolantSystem* class, and so on. The class hierarchy above is showed in the class diagram, Fig.1. In other words, after the basic classes from the given model are composed, a large scale class can be constructed using those. Then, the functions or methods can be logically written using the already designed class. Consequently, *Reactor* as the higher level class can own the all variables of the lower level class, *FuelAssembly*, etc.

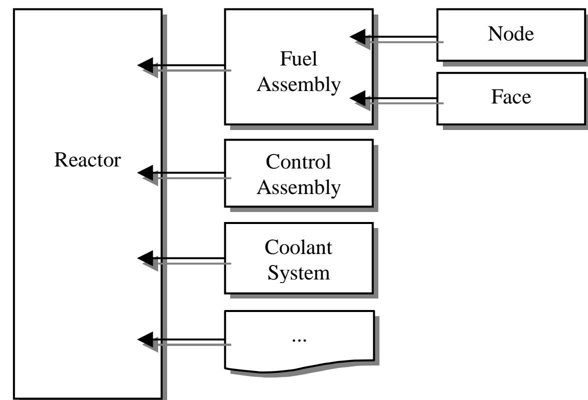

Fig. 1. Class hierarchy in neutronics code

2.2 Variables and Methods

As mentioned above, *Node* class has all variables, node average flux and cross sections, etc., that are contained by elementary mesh of the neutronics solver. For example, *Node* class can be written by using *Type* as the followings.

```
type Node
    type(CrossSection), pointer :: xs
    type(Face), dimension(:) :: faces
    real     ::  size
    real, dimension(:), pointer :: flux_avg
    ……
end type Node
```

Face class can be designed by the same manner and it has variables as like partial currents, face flux and boundary information. Then, *FuelAssembly* class can be composed of *Node* and *Face* class. Eventually, *Reactor* class has the variables of *FuelAssembly Type* and *ControlSystem Type*, and so on, as the followings.

```
type FuelAssembly
    real ::power
    type(Node), dimension(:), pointer :: nodes
    type(Face), dimension(:), pointer :: faces
    ......
end type FuelAssembly

type Reactor
    type(FuelAssembly), dimension(:), pointer :: fuel_assm
    type(CoolantSystem), dimension(:),pointer  :: coolant_sys
    ……
end type Reactor
```

Since *Module* of Fortran 90/95 can contain subroutines or functions as the same notation with the method in OOP, the each class has its own methods inside *Module*. This feature adds high modularity to a code. For example, constructor and copy constructor functions for *Reactor* class can be written as the following source. Especially, the copy constructor of *Reactor* class was useful in the reactivity calculations which involve changing thermal hydraulic conditions.

```
function newReactor( n )  result( rx )
     type(Reactor) :: rx
     integer(NBI), intent(in) :: n

     rx%no_mesh = n
     allocate( rx%coolant_system(n) )
     allocate( rx%fuel_assm(n) )
     ......
end function newReactor

function copyReactor( rx ) result( new_rx )
     type(Reactor) :: rx, new_rx
     new_rx%no_axial_mesh = rx%no_axial_mesh
     new_rx%fuel_assm(:) = copyFuelAssembly( rx%fuel_assm(:) )
     ......
end function copyReactor
```

2.3 Data Passing

Because the well designed classes using the OOP features has independence to other classes, the member data inside a class can be efficiently transferred to other classes or functions and can be easily exported to other codes. The following source shows two examples written by old manner and by new type.

```
!! Old Type Data Passing
call solveSystem( flux[], sigma[], current[] , size[], float k_eff, .... )

!! New Type Data Passing
call solveSystem( reactor )
```

All data in the old type source have to be individually transferred to external functions through the arguments or the form of common variables. But, the only argument in the program of new type is a single *Reactor* which contains all data such as *FuelAssembly* and *CoolantSystem*. Therefore, this structure has the advantages of the high portability and the better readability.

3. Data Structure

3.1 Data Type for Pointer Array

Pointer in Fortran 90/95 only means alias and cannot store memory address such as C or C++. Hence, an array storing pointer data type is not declared in Fortran. In order to overcome the weakness, intermediate user-defined data type storing only one pointer variable can be defined as the following source code.

Then, *RealPointer Type* variable can be declared as array in other *Type* or *Module* and this alternate data type can carry on the same function with the pointer array in C++.

```
type RealPointer
     real(NBR), pointer :: ptr
end type RealPointer

type Face
     type(RealPointer), dimension(:,:), pointer :: node_flux
end type Face
```

3.2 Linked List

Node as mentioned in the previous chapter has two *Face*s in both sides per one direction and one *Face* has two neighboring *Node*s. Though *Node* and *Face* are independent class, flux in *Node* and currents in *Face* obtained by nodal solver should be simultaneously updated and readily affect to neighboring *Node* or *Face*. Therefore, similar data structure with the linked list was constructed using pointer array previously described and *Node* and *Face* can be linked as the followings.

```
type Face
     …...
end type Face

type NeighborFace
     type(Face), pointer :: ptr
end type NeighborFace

type Node
     type(NeighborFace), dimension(2) :: faces
end type Node
```

4. Conclusions

The neutronics analysis code, ASTRA1D, including diffusion equation nodal solver was developed using the concept of object oriented programming via Fortran 90/95. The physical reactor model was separated to the abstract elements and redefined by the basic classes such as *Node* and *Face*, according to the principal of OOP. Then, they were integrated into *FuelAssembly* class and *Reactor* class. In addition, constructor and copy constructor for user-defined data type were newly defined and pointer array and linked list were designed.

Consequently, the data structure and the functions has the modularity and the portability known as the features of OOP and the entire neutronics analysis code could be efficiently developed and worked.

ACKNOWLEDGMENTS

This work was supported by the Nuclear Research & Development of the Korea Institute of Energy Technology Evaluation and Planning (KETEP) grant funded by the Korea government Ministry of Knowledge Economy.

REFERENCES

[1] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley Pub. Co., 3rd Edition, (2000).
[2] Ed Akin, Object Oriented Programming via Fortran 90/95, Rice Univ., Texas, USA (2001).