OBJECT ORIENTED PROGRAMMING IN FORTRAN

André Teófilo Beck, atbeck@sc.usp.br
Felipe Alexander Vargas Bazán, favb@sc.usp.br
Department of Structural Engineering, São Carlos School of Engineering, University of São Paulo, Av. Trabalhador Sãocarlense 400, 13566-590, São Carlos, SP, Brazil

Abstract. *This paper presents fundamental concepts for object oriented programming (OOP) in FORTRAN. In general, FORTRAN users are not familiar with these concepts since, until recently, there was no support for object oriented programming in FORTRAN compilers. Recently, version 11.1 of the Intel Visual Fortran compiler was released, incorporating support for most object oriented features of the FORTRAN 2003 standard. Hence, FORTRAN users can now update their practice with this important programming methodology. The main purpose of this paper is to show that FORTRAN can be used with a level of abstraction much higher than observed in practice (in particular, in engineering), by using OOP concepts. The article presents the state of the art in FORTRAN compilers and standards, with respect to OOP. The article discusses the concepts of data abstraction, encapsulation and information hiding, classes and objects. Concepts are presented independently of implementation language, but their implementation is illustrated in FORTRAN 90/95/2003. The article illustrates construction of polymorphic classes, by type-extension and inheritance, using the Intel Visual FORTRAN compiler version 11.1. The article also shows that polymorphism can be emulated, by the appropriate use of pointers, using older compilers and FORTRAN 90/95. Implementation of concepts is illustrated by means of an academic and didactic example, involving a university management system, which manipulates persons, students, professors, courses and dates.*

Keywords*: computer programming, object oriented programming, OO programming, FORTRAN*

1. INTRODUCTION

This paper aims to disseminate concepts of object oriented programming (OOP) for FORTRAN language users. Such concepts will make it possible to take best advantage of resources of OOP introduced by FORTRAN 2003 standard and recently implemented in Intel Visual FORTRAN compiler version 11.1 (Intel, 2009). The article shows that the language can be used with a level of abstraction much higher than observed in practice in the scientific environment (in particular, in engineering).

When it was created by John Backus in the 50's, FORTRAN – FORmula TRANslator – was considered as a high-level language, in comparison with other machine languages of that time (e.g., ASSEMBLER). Thus, FORTRAN had a fast dissemination and gained thousands of users. FORTRAN played a fundamental role in making computer programming accessible to the scientific community. In consequence, a large number of programs was developed in this language, over many years. FORTRAN is, until today, extremely popular among scientists.

Since its creation, FORTRAN users developed several "dialects" of the language. With the objective of standardizing the language, in 1966 the first standard of a programming language was created, namely, FORTRAN 66. Later, in 1977, a new standard (FORTRAN 77) was created to standardize the language, incorporating some of the many resources developed independently by different groups. During this development, FORTRAN always exploited hardware resources very well, making it, until these days, one of the fastest programming languages..

In the decades 60 to 80, new programming languages appeared. Several of them surpassed FORTRAN in terms of resources, mainly those resources enabling a more abstract programming level. The FORTRAN 90 standard (Adams *et al.*, 1992) was created not to standardize existing practices, but to incorporate in FORTRAN resources already present in other languages. This means that the standard (the paper) began to specify resources to be implemented in compilers. The FORTRAN 90 standard was a major revision. Some of the main new features introduced to the language were: operations with matrices, many intrinsic functions, dynamic matrix allocation, user-defined data types, modules, module procedures, operator overloading, interfaces, operations with pointers, and free-form programming. Many of these resources are not extensively employed, for example, by engineering students programming in FORTRAN. In this article, some of these resources, needed to implement OOP, are addressed. The FORTRAN 95 standard (Adams *et al.*, 1997) did not incorporate great resources to the language, but made it compatible with HPF – High Performance FORTRAN – for parallel processing.

The FORTRAN 2003 standard (Adams *et al.*, 2009) introduced several concepts of OO programming, such as type extension, the CLASS keyword, resources for inheritance and polymorphism, and type-bound procedures, as described in this paper. During many years, this standard was ahead of the compilers, since until very recently no FORTRAN compilers did support most of the resources introduced by FORTRAN 2003. In October 2009, Intel released the Intel Visual FORTRAN compiler version 11.1 (Intel, 2009), which offers support for a large part (but not all) of the object orientation resources of FORTRAN 2003. Until then, some OO programming techniques could be emulated in older compilers using FORTRAN 90/95 resources (Akin, 2003; Decyk *et al.*, 1997; Decyk *et al.*, 1998).

This article is organized as follows. Section 2 presents definition, objectives and components of OO programming. Section 3 describes the concept of data abstraction. In Section 4 the concepts of class and object are presented. In Section 5 construction of classes by composition, using FORTRAN 90/95/2003, is illustrated. Section 6 presents construction of polymorphic classes by using inheritance according to FORTRAN 2003 standard. Section 7 shows how polymorphic classes can be created by emulation, using the FORTRAN 90/95 standard. In Section 8 some conclusions are presented.

Since the article deals with compiler instructions and code excerpts, the following syntax is used:

`CODE INSTRUCTIONS` – programming instructions and program excerpts appear exclusively in capital letter and Courier New reduced font (size 7).

`LARGE CODE PORTIONS` – large program portions appear within a chart, with a corresponding figure numeration. Programming instructions follow the syntax above, with capital letters and Courier New reduced font.

2. OO PROGRAMMING: DEFINITION, OBJECTIVES AND COMPONENTS

The philosophy of object orientation is based on the construction of software as a structured collection of classes. Focus of the development is not on the tasks which the software must execute, but on the objects (from the real world or not) which the software must handle. In other words, the focus of the development is not on what the program does, but on which objects it manipulates.

The main objectives of OO programming are expansibility, reusability, and compatibility. Expansibility is the capability of adapting or extending a piece of software due to changes in specifications. Reusability is the capability of using programs or software elements which have been previously programmed, in order to construct new applications. Compatibility is the capability of combining together software elements developed by different parts. In addition to these objectives, other internal factors (only realized by those involved in code development) are code legibility and capability of developing large programs in a logical and objective manner.

Object oriented programming has four fundamental components: data abstraction, classes, inheritance, and polymorphism. Data abstraction consists of extracting the essential features of the objects to be manipulated by the program. A class consists of the implementation of such features in an independent program unit, equipped with the data structure and the functions needed to manipulate objects. An object is a portion of information created (or instantiated) from a class at run-time. Development of the basic unit (the class) is based on encapsulation and information hiding: essential aspects of the object behavior are visible from outside the class, but implementation details remain hidden. Inheritance is the property by which specific (sub-)classes can be created by inheriting features from more general classes. Inheritance gives rise to families of classes and to polymorphism. Polymorphism is the capability of developing software which manipulates, generically, any object originated from a family of classes. This includes construction of software which is able to manipulate objects instantiated from classes developed later than the actual software.

3. DATA ABSTRACTION (ABSTRACT DATA TYPES)

A fundamental aspect in OO programming is the identification and characterization of the objects (from the real world or not) which the program to be developed has to manipulate. Data abstraction is the capability of extracting the essential features of these objects. The features are defined from behavior; implementation is secondary and remains hidden. Description of the objects must be such that any user is capable of using parts of the program without knowing internal details about how they were programmed. The term Abstrac Data Type (ADT) is employed, but this nomenclature does not correspond to a data type of FORTRAN compiler. The concept of data abstraction is independent from language, so it can be applied to any programming language.

Proper description of an ADT has to be precise and unambiguous, has to be as complete as needed, and should not be over-specifying (Meyer, 2000). Focus is placed on the operations to be executed on the object. In order to perform a specification which is independent from implementation, it is suitable to employ a system of signatures, as illustrated in the following example.

3.1. Specification of an ADT `CALENDAR_DATE`

Any program which manipulates dates can use an ADT `CALENDAR_DATE`. Description of this ADT must be such that its use does not depend on the data system chosen by the user, i.e., it has to work properly either in system DAY/MONTH/YEAR or in the system MONTH/DAY/YEAR. The example is simple, but serves the purpose of illustrating the concept.

a. Type specification: `TYPE(CALENDAR_DATE) DATE`
b. List of functions:

```
For any I:integer, R:real, L:logical and DATE:TYPE(CALENDAR_DATE)
CREATE:      → DATE
```

```
SET_DAY:                   DATE x I → DATE      GET_DAY:      DATE → I
SET_MONTH:                 DATE x I → DATE      GET_MONTH:    DATE → I
SET_YEAR:                  DATE x I → DATE      GET_YEAR:     DATE → I
DATE1_LESS_THAN_DATE2:     DATE x DATE → L
DATE1_GREATER_THAN_DATE2: DATE x DATE → L
DATE_DIFFERENCE:           DATE x DATE → DATE
CONVERT_TO_DAYS:           DATE → I
PRINT_DATE_DMY_FORMAT:     DATE → Ø
PRINT_DATE_MDY_FORMAT:     DATE → Ø
```

c. Axioms:

```
For any I:integer, R:real, L:logical and DATE:TYPE(CALENDAR_DATE)
DATE = CREATE_DATE()
SET_DAY(DATE,I)   = DATE                  GET_DAY(DATE)   = I
SET_MONTH(DATE,I) = DATE                  GET_MONTH(DATE) = I
SET_YEAR(DATE,I)  = DATE                  GET_YEAR(DATE)  = I
DATE1_LESS_THAN_DATE2(DATE,DATE) = L
DATE1_GREATER_THAN_DATE2(DATE,DATE) = L
DATE_DIFFERENCE (DATE,DATE) = DATE
CONVERT_TO_YEARS(DATE) = R
PRINT_DATE_DMY_FORMAT(DATE)
PRINT_DATE_MDY_FORMAT(DATE)
```

d. Pre-conditioners:

```
SET_DAY(DATE,I)     requires: 1 ≤ I ≤ number of days of each month
SET_MONTH(DATE,I)   requires: 1 ≤ I ≤ 12
SET_YEAR(DATE,I)    requires: 0 ≤ I  ?
```

In the list of functions (sub-section b. above) a nomenclature of signatures is used, which is independent from programming language and similar to that used for mathematical functions. In the list above, functions where `DATE` appears on the right of the arrow are creation functions, i.e., they produce an instance of `DATE` from another type or from no argument. When `DATE` appears on the left of the arrow, the function is an inquiry function (query), which returns properties of one (or more) instance(s) of `DATE`. Functions in which `DATE` appears on both sides of the arrow are command (action) functions, which modify a certain instance of `DATE`.

It can be observed that the description above is to work correctly, no matter the date format the user is thinking, i.e., DAY/MONTH/YEAR or MONTH/DAY/YEAR. As a counter-example, any function that permits simultaneous specification of the three parameters, e.g., `CREATE_DATE(DAY,MONTH,YEAR)=DATE`, would give rise to run-time errors if parameters DAY/MONTH were interchanged.

3.2. Implementation of DATA TYPES in FORTRAN

The concept of data abstraction presented above is, by definition, independent of programming language. For purposes of illustration, we present in this section one possible implementation of ADT `CALENDAR_DATE` in FORTRAN.

ADTs can be implemented in FORTRAN by use of *user-defined data types*. This resource allows creation of data structures more complex than the so-called *intrinsic data types*, i.e., integer, real, double-precision real, logical variables, etc. In FORTRAN syntax, one possible implementation of ADT `CALENDAR_DATE` is:

```
TYPE CALENDAR_DATE
   INTEGER :: DAY=1, MONTH=1, YEAR=1
END TYPE CALENDAR_DATE
```

Variables `DAY`, `MONTH` and `YEAR` are called the components of the data type. Declaration of a data type `CALENDAR_DATE` in the main program or in a subroutine creates an instance of this data type. A variable to store the date of birth of a person, e.g., is created with the statement:

```
TYPE(CALENDAR_DATE) :: BIRTH_DATE
```

In FORTRAN, components of a data type are accessed by the character %. Thus, the year of birth of this person is `BIRTH_DATE%YEAR`. A vector to store the date of birth of 10 people is created as:

```
TYPE(CALENDAR_DATE) :: BIRTH_DATE(10)
```

The eighth component of vector `BIRTH_DATE` is accessed as `BIRTH_DATE(8)`. The year of birth of this person is `BIRTH_DATE(8)%YEAR`. The day of birth of all the 10 people is accessed with the syntax `BIRTH_DATE(1:10)%DAY`. Vector `BIRTH_DATE` can also be created with allocatable dimensions:

```
TYPE(CALENDAR_DATE),ALLOCATABLE :: BIRTH_DATE(:)
```

The components of a data structure can also be vectors. In Compaq Visual FORTRAN compiler, version 6.6.0 and later, these components can also be allocatable. As an example, consider a data structure to store information about a

person, like names and date of birth. To store strings with the exact number of characters needed, this data type could be written as:

```
TYPE PERSON
   CHARACTER, ALLOCATABLE :: NAME(:),MIDDLENAME(:),SURNAME(:)
END TYPE PERSON
```

An instance of this data type is created with the statement:

```
TYPE(PERSON) :: PROFESSOR
```

After the number of characters of each name is known (by reading a temporary string, for example), the allocation statement is executed for each component:

```
ALLOCATE(PROFESSOR%NAME(5),PROFESSOR%MIDDLENAME(7),PROFESSOR%SURNAME(4))
```

Components can be allocated within a creation function, belonging to a class, as will be shown later. After the components are allocated, values may be assigned to them:

```
PROFESSOR%NAME(1:5)       = (/'A','N','D','R','E'/)
PROFESSOR%SURNAME(1:4)    = (/'B','E','C','K'/)
```

Data structures can also have allocatable dimensions. An allocatable vector of students is created with the statement:

```
TYPE(PERSON), ALLOCATABLE :: STUDENT(:)
```

Thirty instances of this data type (i.e., a vector of 30 students) are created with the statement:

```
ALLOCATE(STUDENT(30))
```

Obviously, this allocation has to occur before allocation of anyone of the components. If the name of the first student has 6 letters, allocation of the component is:

```
ALLOCATE(STUDENT(1)%NAME(6))
```

The clear and concise syntax of a data structure helps to significantly improve code legibility, as well as to construct complex programs at a high level of abstraction.

3.3. Operator overloading

Operator overloading allows standard operators for intrinsic data types (`+` , `-` , `<` , `>` , `≤` , `==` , `.EQ.` , `.LT.` , etc.) to be extended also for more complex data structures. As an example, in order to sort a set of dates in chronological order, or simply to compare the chronological order of two dates, operators < and > may be overloaded in such a way as to operate on `CALENDAR_DATE` data types. A function which performs the comparison between the dates, and returns a logical variable with the result of this comparison, is created first. A function to perform the comparison `DATE1 < DATE2` is illustrated in Fig. 1. Operator overloading is done using the following statement:

```
INTERFACE OPERATOR (.LT.)
   MODULE PROCEDURE DATE1_LESS_THAN_DATE2
END INTERFACE
```

This declaration must be contained within a module, as illustrated in Fig. 3.

```
!------------------------------------------------------------------------
! VERIFIES IF DATA1 IS LESS THAN DATA2. RESULT IS A LOGICAL VARIABLE (T OR F)
!------------------------------------------------------------------------
FUNCTION DATE1_LESS_THAN_DATE2(DATE1,DATE2) RESULT(LESS)
   TYPE(CALENDAR_DATE), INTENT(IN) :: DATE1,DATE2
   LOGICAL :: LESS = .FALSE.
   IF(DATE1%YEAR.LT.DATE2%YEAR) THEN
      LESS = .TRUE.
   ELSE IF ((DATE1%YEAR.EQ.DATE2%YEAR).AND.(DATE1%MONTH.LT.DATE2%MONTH)) THEN
      LESS = .TRUE.
   ELSE IF ((DATE1%YEAR.EQ.DATE2%YEAR).AND.(DATE1%MONTH.EQ.DATE2%MONTH).AND. &
                                          (DATE1%DAY.LT.DATE2%DAY)) THEN
      LESS = .TRUE.
   ENDIF
END FUNCTION DATE1_LESS_THAN_DATE2
```

Figure 1. FORTRAN code illustrating a function to compare two dates, overloaded with operator < (less than).

4. CLASSES AND OBJECTS

4.1. Classes

Following Meyer (2000), the implementation of an ADT through a particular form of representation and the codification in computer language is known as a class. Classes are constructed in such a way as to encapsulate all variables, functions and subroutines needed to manipulate an object. In FORTRAN, encapsulation is performed by using modules. Modules are the only tool available in FORTRAN to create blocks of statements isolated from the rest of the program.

Every class must have a public part and a secret part, as illustrated in Fig. 2. The public part, which must be known by the users of the class, corresponds to the specifications of the ADT. The secret part corresponds to the particular choice of representation and implementation chosen by the programmer. Based on the public part and on the correct identification of the ADT features, any user can use a class without knowing the specific implementation details. In FORTRAN, by default, components of a user-defined data type, as well as functions and subroutines of a class, are public, unless otherwise specified. The concept of data hiding, fundamental in OO programming, implies that data type components should be declared as private. Only functions and subroutines (but not all) should be public. Figure 3 illustrates an implementation of the class `CALENDAR_DATE` with private components.

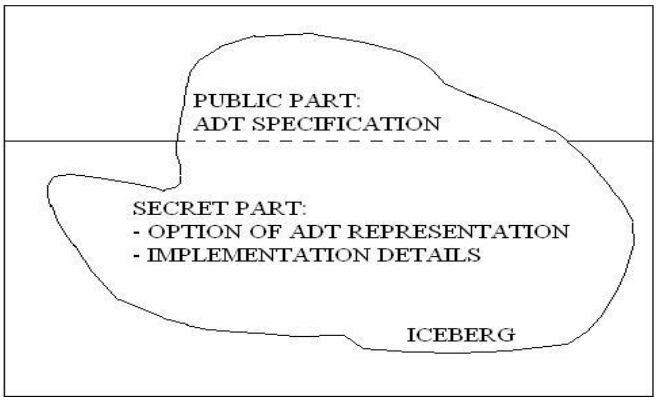

Figure 2. Illustration of public and secret parts of a class.

When the components of a data type are private, they cannot be accessed from outside the class. Thus, any part of the main program which uses `TYPE(CALENDAR_DATE)` will not be able to directly access the day through the statement `DATE%DAY`. This means that specific functions must be created with this objective, such as the function `GET_DAY(DATE)=DAY`, illustrated in Fig. 3. This implies the programming of several functions which, in principle, could appear to be unnecessary. However, these same functions will later allow polymorphism of the class.

```
!-------------------------------------------------------------------------------
!   IMPLEMENTATION OF AN ABSTRACT DATA TYPE CALENDAR_DATE IN FORTRAN 90/95/2003
!-------------------------------------------------------------------------------
MODULE CLASS_CALENDAR_DATE
   IMPLICIT NONE
   TYPE, PUBLIC :: CALENDAR_DATE
      PRIVATE
      INTEGER :: DAY=1, MONTH=1, YEAR=0001
   END TYPE CALENDAR_DATE
   INTEGER, PARAMETER :: DAYS_PER_MONTH(12) = (/31,29,31,30,31,30,31,31,30,31,30,31/)
!-------------------------------------------------------------------------------
!  OVERLOADING OPERATOR 'LESS THAN' OR .LT. TO COMPARE DATES
!-------------------------------------------------------------------------------
   INTERFACE OPERATOR (.LT.)
      MODULE PROCEDURE DATE1_LESS_THAN_DATE2
   END INTERFACE
                    ! OTHER OPERATOR OVERLOADING DEFINITIONS FOLLOW
!-------------------------------------------------------------------------------
CONTAINS
!-------------------------------------------------------------------------------
!  SET_DAY(CD,I) ! FUNCTION WHICH VERIFIES AMD ASSIGNS A DAY NUMBER TO A DATE
!-------------------------------------------------------------------------------
   SUBROUTINE SET_DAY(CD,X)
      INTEGER,INTENT(IN) :: X                     ! DAY TO BE ASSIGNED
      TYPE(CALENDAR_DATE),INTENT(INOUT) :: CD     ! DATE
      IF(X.LT.1 .OR. X.GT.DAYS_PER_MONTH(CD%MONTH)) THEN
            PRINT *, 'INVALID DAY!'
      ELSE
            CD%DAY = X
      ENDIF
   END SUBROUTINE SET_DAY
```

```
!-------------------------------------------------------------------------------
!  I = GET_DAY(CD) ! FUNCTION WHICH RETURNS THE DAY OF A DATE OBJECT
!-------------------------------------------------------------------------------
   FUNCTION GET_DAY(CD) RESULT(X)
      INTEGER X                                ! DAY
      TYPE(CALENDAR_DATE),INTENT(IN) :: CD     ! DATE
      X = CD%DAY
   END FUNCTION GET_DAY
!-------------------------------------------------------------------------------
!  DATE1_LESS_THAN_DATE2(DATE,DATE) = L
!-------------------------------------------------------------------------------
   FUNCTION DATE1_LESS_THAN_DATE2(DATE1,DATE2) RESULT(LESS)
                     ! IMPLEMENTATION AS IN FIG. 1
   END FUNCTION DATE1_LESS_THAN_DATE2
                     ! OTHER FUNCTIONS AND SUBROUTINES
END MODULE CLASS_CALENDAR_DATE
```

Figure 3. Implementation of class CALENDAR_DATE in FORTRAN 90/95/2003.

4.2. Objects

Objects are instances of classes, created at run-time from the molds (the classes themselves). Therefore, all the OO programming is, in fact, based on ADTs and classes; objects are only created (or instantiated) at run-time. The relationship between ADTs, classes and objects is illustrated in Fig. 4. The objects instantiated at run-time are the same objects characterized trough the abstract data type (ADT) definition.

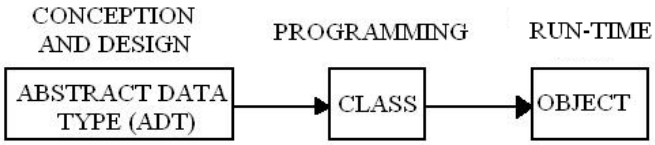

Figure 4. “Temporal” relationship between ADTs, classes and objects.

5. CONSTRUCTION OF CLASSES BY COMPOSITION

There are two ways to construct (more complex) classes from other classes: composition and inheritance. Inheritance is described in Section 6.

In composition, an existing class is employed in the construction of a new class. Thus, the new class uses all the components and methods of the existing class, as if it were a client of the latter. As an example, Fig. 5 illustrates the construction of a class PERSON using the data type CALENDAR_DATE, whose implementation was illustrated in Fig. 3.

```
!-------------------------------------------------------------------------------
!   CLASS PERSON - IMPLEMENTATION OF A CLASS PERSON (FORTRAN 90/95/2003)
!-------------------------------------------------------------------------------
MODULE CLASS_PERSON
USE CLASS_CALENDAR_DATE
IMPLICIT NONE
!-------------------------------------------------------------------------------
   INTEGER, PARAMETER :: NS = 20
   CHARACTER(NS), PARAMETER :: BLANK = '                    '
   TYPE PERSON
      PRIVATE
      CHARACTER(NS) ::       NAME = BLANK     ! FIRST NAME
      CHARACTER(NS) :: MIDDLENAME = BLANK     ! MIDDLE NAME
      CHARACTER(NS) ::    SURNAME = BLANK     ! FAMILY NAME
      INTEGER       :: CPF = 0          ! PERSONAL REGISTRATION NUMBER (SOCIAL SECURITY)
      INTEGER       ::  RG = 0          ! PERSONAL IDENTIFICATION NUMBER
      INTEGER       :: AGE = 0          ! AGE
      TYPE(CALENDAR_DATE) :: DOB        ! DATE OF BIRTH
   END TYPE PERSON
!-------------------------------------------------------------------------------
CONTAINS
!-------------------------------------------------------------------------------
!  UPDATE_AGE(PE) - DETERMINES THE AGE OF A PERSON FROM THE CURRENT DATE
!-------------------------------------------------------------------------------
   SUBROUTINE UPDATE_AGE(PE)
      USE DFLIB
      TYPE(PERSON)         PE
      TYPE(CALENDAR_DATE) TODAY
      INTEGER(2)           D,M,Y
      CALL GETDAT(Y,M,D)  ! FUNCTION OF LIBRARY DFLIB RETURNS CURRENT YEAR, MONTH AND DAY
      CALL SET_YEAR(TODAY,Y)
      CALL SET_MONTH(TODAY,M)
      CALL SET_DAY(TODAY,D)
      PE%AGE = INT(CONVERT_TO_YEARS(TODAY-PE%DOB))
   END SUBROUTINE UPDATE_AGE
```

```
!-------------------------------------------------------------------------------
!  C = GET_PERSON_FULL_NAME(PE) ! RETURNS A STRING WITH THE FULL NAME
!-------------------------------------------------------------------------------
   FUNCTION GET_PERSON_FULL_NAME(PE) RESULT(FULL_NAME)
      TYPE(PERSON) PE
      CHARACTER(3*NS)   FULL_NAME
      IF(PE%MIDDLENAME.EQ.BLANK) THEN
         FULL_NAME = TRIM(PE%NAME)//' '//TRIM(PE%SURNAME)//BLANK//BLANK
      ELSE
         FULL_NAME = TRIM(PE%NAME)//' '//TRIM(PE%MIDDLENAME)//' '//TRIM(PE%SURNAME)
      ENDIF
   END FUNCTION GET_PERSON_FULL_NAME
!-------------------------------------------------------------------------------
!  I = GET_PERSON_ID(PE)        ! RETURNS ID NUMBER (RG) OF A PERSON
!-------------------------------------------------------------------------------
   FUNCTION GET_PERSON_ID(PE) RESULT(ID)
      TYPE(PERSON) PE
      INTEGER ID
      ID = PE%RG
   END FUNCTION GET_PERSON_ID
END MODULE CLASS_PERSON
```

Figure 5. Illustration of composition: construction of class PERSON using class CALENDAR_DATE.

In the example illustrated in Fig. 5, class PERSON is a client of class CALENDAR_DATE, since it uses methods of the latter to manipulate dates. The statement USE_CLASS_CALENDAR_DATE allows using elements of this class. To compare the age of two persons, class PERSON can employ directly the function DATE1_LESS_THAN_DATE2 or the overloaded operator "<" (illustrated in Fig. 3). The date of birth of a person is defined by using functions SET_ of class CALENDAR_DATE (Fig. 3).

In the example, the function which updates the age of a person, UPDATE_AGE, employs functions of the class CALENDAR_DATE, in the bold face statement (Fig. 5). One of these is the function *difference between dates*, overloaded on the subtraction operator "–" and which also results in a data type *date* (difference in years, months and days). The other function is CONVERT_TO_YEARS, which converts a date (in fact, the difference between two dates) in years. The syntax becomes extremely legible: the statement in bold face shows explicitly that the age of a person is calculated in integer years, from the difference between the current date and the date of birth of this person.

Functions GET_PERSON_FULL_NAME and GET_PERSON_ID, showed in Fig. 5 for purposes of illustration, are used later in this article.

6. INHERITANCE AND POLYMORPHISM IN FORTRAN 2003

One of the most important resources in an OO language is inheritance (Meyer, 2000). With inheritance, it is possible to create derived classes (offsprings) which inherit their main features from the base class (the parent class). OO languages are able to automatically create polymorphic classes, based on the extension of data types. Polymorphism allows a program to manipulate, in a generic form, objects instantiated from different classes, which have been (or will be) generated from different extensions of the same general class. Polymorphism is fundamental to obtain expansibility and reusability of parts of the program.

Inheritance and polymorphism were incorporated in FORTRAN through the FORTRAN 2003 standard. However, until recently there were no commercial compilers with support for these language resources. Only in October 2009, Intel Visual FORTRAN compiler version 11.1 (Intel, 2009) was released, supporting inheritance and polymorphism. The example presented in this section must be run in this compiler or in a later version.

Consider the specification of the class PERSON in Fig. 5. To construct an university management program, it is important to distinguish between two fundamental types of persons: professors and students. Both are persons, hence both have name, surname, identity documents, date of birth, etc. However, professors and students are particular types of persons, at least within the university environment. A student has associated to him/her, for example: a program (e.g., graduate in civil engineering), a scholar identification number, a date of enrolment in the program, a date of exit (graduation), and a school transcript containing information about courses such as name, year, semester and final grade. On the other hand, a professor has, associated to him/her, a date of hiring, the monthly salary, a bank account, information about his/her dependents, etc.

Professors and students are particular types of persons, hence the classes STUDENT and PROFESSOR can be constructed based on class PERSON (parent class). Figure 6 illustrates the construction of class STUDENT by inheritance, in FORTRAN 2003, using the keyword EXTENDS. When class PERSON is "extended", class STUDENT inherits its components and methods. Therefore, objects of class STUDENT also have components NAME, MIDDLENAME, SURNAME, AGE, etc. The age of a student, e.g., is the component STUDENT%AGE. In a similar manner, as illustrated in Fig. 6, a class PROFESSOR can be constructed.

```
!-------------------------------------------------------------------------------
! IMPLEMENTATION OF CLASS STUDENT IN FORTRAN 2003 (INTEL VISUAL FORTRAN V. 11.1.048).
! BY INHERITANCE, TYPE STUDENT INHERITS COMPONENTS AND METHODS FROM CLASS PERSON.
! BY COMPOSITION, TYPE STUDENT USES COMPONENTS AND METHODS OF CLASS CALENDAR_DATE.
!-------------------------------------------------------------------------------
MODULE CLASS_STUDENT
   USE CLASS_PERSON
```

```
!  USE CLASS_TRACK_RECORD ! CLASSES IMPLEMENTING SCHOOL TRANSCRIPT FIELDS SUCH AS
!  YEAR/SEMESTER, FREQUENCE, OBTAINED NOTE, FINAL RESULT (APPROVED OR REPPROVED)
   IMPLICIT NONE
   TYPE, PUBLIC, EXTENDS(PERSON) :: STUDENT
      PRIVATE
      INTEGER              :: ID = 1       ! NUMBER OF SCHOLAR IDENTITY
      TYPE(CALENDAR_DATE) :: ENROLMENT    ! DATE OF ENROLMENT IN THE PROGRAM
      TYPE(CALENDAR_DATE) :: CONCLUSION   ! DATE OF COMPLETION OF THE PROGRAM
!     TYPE(TRACK_RECORD)  :: TR(50)       ! HISTORY OF UP TO 50 COURSED MATTERS
      CONTAINS
         PROCEDURE,PUBLIC :: GET_STUDENT_ID
   END TYPE STUDENT
!-------------------------------------------------------------------------------
   CONTAINS
!-------------------------------------------------------------------------------
!  I = GET_STUDENT_ID(ST)  ! RETURNS NUMBER OF SCHOLAR IDENTITY OF THE STUDENT
!-------------------------------------------------------------------------------
   INTEGER FUNCTION GET_STUDENT_ID(ST) RESULT(ID)
      TYPE(STUDENT) :: ST
      ID = ST%ID
   END FUNCTION GET_STUDENT_ID
END MODULE CLASS_STUDENT
```

Figure 6. Implementation of class `STUDENT` by type extension using FORTRAN 2003.

The complete dimension of the power of a class is given by the polymorphism obtained from *type extensions*. A program can be written to manipulate persons, in a generic form, no matter if they are students, professors or dependents of these. Certainly, there are some functions which are specific to each of the specialized classes. Following the example, only students receive grades, and only professors receive salaries. However, still as an example, consider a program for library management. The library lends books to students, professors, or dependents of these (persons in general). Hence, a user of the library may belong to any of the discussed classes. In the library management program, a user can be declared as:

```
CLASS(PERSON),POINTER :: USER
```

This FORTRAN syntax makes clear that `USER` is a pointer. The `CLASS` keyword, in this case, indicates that this pointer will be able to point to objects of class `PERSON` or of any of its derived classes. Therefore, declaring the possible targets of this pointer:

```
TYPE(PERSON ),   TARGET :: PE
TYPE(STUDENT),   TARGET :: ST
TYPE(PROFESSOR), TARGET :: PR
```

the program will be able, at run-time, to decide which type of library users the pointer `USER` has to point to. For example, `USER=>ST` creates, at run-time, an association to an object of class `STUDENT`. `PERSON` is said to be the declared data type of the variable `USER`, and `STUDENT` and `PROFESSOR` are said to be the dynamic data types.

The library management program is written in such a way as to generically manipulate users, no matter if they are `PERSON`, `STUDENT`, `PROFESSOR` or even a data type created a posteriori (e.g., an employee). For this purpose, a small "correction" is needed in the code presented in Fig. 5 (which is written in FORTRAN 90/95 standard): in the definition of the functions of class `PERSON`, the statement `TYPE(PERSON)` has to be replaced by `CLASS(PERSON)`. This allows all the derived classes to use functions of the parent class (`UPDATE_AGE`, `GET_PERSON_FULL_NAME`, `GET_PERSON_ID`, etc.). Therefore, the full name of the generic user `USER` above is obtained with the statement `GET_PERSON_FULL_NAME(USER)`. For the Intel version 11.1 FORTRAN compiler, functions `UPDATE_AGE` and `GET_PERSON_FULL_NAME` are automatically polymorphic, since they can operate with arguments of class `PERSON` or of any of the derived sub-classes.

Functions of the parent class which operate with arguments `CLASS(PERSON)` can be directly used by the derived classes. However, in specific situations, it could be necessary to redefine the behavior of certain functions for some of the derived classes. A student, for example, has two personal identifications (RG and CPF) and a university identification number. For an university management program, the student identification number could be more relevant than the RG or CPF. Therefore, it could be necessary to reformulate function `GET_PERSON_ID`, such that it returns the the student identification number if the argument is from class `STUDENT`. This is known as procedure overriding. When a derived sub-class is created by type extension, some functions are rewritten, using the same name but operating exclusively with arguments of the derived class. This procedure is already foreseen in FORTRAN 2003 standard (Chapman, 2007) but, apparently, it is not yet supported by Intel Visual Fortran compiler version 11.1. It is supposed that this resource will be available soon, in new versions of the compiler. In the meantime, this compiler limitation can be overcome by using the construct `SELECT_TYPE` (only FORTRAN 2003). Figure 7 illustrates this procedure the function `GET_PERSON_ID`. Statement `SELECT TYPE` is able to identify the data type of the argument, making explicitly the selection of the proper function to be used for each argument. It is noted that, in this case, the function `GET_PERSON_ID` has to be rewritten, with different names, for each one of the possible arguments. If the *procedure overriding* construct were

available, the functions would be rewritten with the same name (e.g., `GET_ID`) and polymorphism would be automatically made by the compiler.

```
!-------------------------------------------------------------------------------
! P = POINTER TO TYPE PERSON OR TO ANY OF ITS DERIVED TYPES.
! FUNCTIONS GET_TYPE AND GET_ID WILL BE ABLE TO BE IMPLEMENTED BY OVERRIDING IN
! LATER VERSIONS OF THE COMPILER.
!-------------------------------------------------------------------------------
MODULE CLASS_PERSON_POINTER
   USE CLASS_PERSON
   USE CLASS_STUDENT
   USE CLASS_PROFESSOR
   IMPLICIT NONE
   TYPE :: PERSON_POINTER
      CLASS(PERSON), POINTER :: P
   END TYPE PERSON_POINTER
!-------------------------------------------------------------------------------
   CONTAINS
!-------------------------------------------------------------------------------
!  C = GET_TYPE(PE) ! RETURNS A STRING IDENTIFYING THE TYPE OF PERSON
!-------------------------------------------------------------------------------
   CHARACTER(LEN=12) FUNCTION GET_PERSON_TYPE(X) RESULT(NAME)
      CLASS(PERSON)   X
      SELECT TYPE(X)
      TYPE IS (PROFESSOR)
         NAME = '   PROFESSOR'
      TYPE IS (STUDENT)
         NAME = '      STUDENT'
      CLASS IS (PERSON)
         NAME = '       PERSON'
      END SELECT
   END FUNCTION GET_PERSON_TYPE
!-------------------------------------------------------------------------------
!  I = GET_ID(PR) ! RETURNS IDENTIFICATION NUMBER OF PERSON, STUDENT, OR PROFESSOR
!-------------------------------------------------------------------------------
   INTEGER FUNCTION GET_ID(X) RESULT(ID)
      CLASS(PERSON) :: X
      SELECT TYPE(X)
      TYPE IS (PROFESSOR)
         ID = GET_PROFESSOR_ID(X)
      TYPE IS (STUDENT)
         ID = GET_STUDENT_ID(X)
      CLASS IS (PERSON)
         ID = GET_PERSON_ID(X)
      END SELECT
   END FUNCTION GET_ID
END MODULE CLASS_PERSON_POINTER
```

Figure 7. Complementation of class `PERSON` using FORTRAN 2003.

Another new feature of FORTRAN 2003 (but not yet available in the Intel compiler v. 11.1) is that methods (functions or subroutines) can also be components of a *user-defined data type*. Methods which are components of a certain class are declared by using the `CONTAINS` keyword within the definition of the type (Fig. 6). In the illustrated example, function `GET_STUDENT_ID` could be accessed by the syntax `I=ST%GET_STUDENT_ID`, in addition to the traditional form `I=GET_STUDENT_ID(ST)`. This functionality, together with the resource of function overriding, would permit the more natural and more compact syntax `I=ST%GET_ID` or `I=PE%GET_ID`. These two functionalities will surely be available soon, in new versions of the Visual Intel FORTRAN compiler.

Figure 8 shows a library management program, constructed to manipulate, generically, persons, professors and/or students. The program employs a vector of persons (`USER`) in order to store information about the users. However, a vector declared as `CLASS(PERSON),POINTER` cannot point to different data types. This limitation is outlined by creation of a *data type* `PERSON_POINTER`, which may point indifferently to objects of class `PERSON` or of any sub-class (Fig. 7). A vector of these objects (`USER(:)`), with allocatable dimension, is created in the main program (Fig. 8). At run-time, the main program obtains the number of users of each type, calculates the total number of users, and allocates the vectors. Then, the program assigns to each component of the vector `USER` one of the available *data types*: `PERSON`, `PROFESSOR` or `STUDENT`, as illustrated. This assignment is made through pointers, i.e., each component of the vector `USER` points to a different *data type*. From this point on, the program acts on the vector `USER` independently from the component *data types*, as long as the function is applicable to class `PERSON`, i.e., to persons in general.

```
!-------------------------------------------------------------------------------
! PROGRAM LIBRARY USERS (FORTRAN 2003, INTEL VISUAL FORTRAN COMPILER V. 11.1.048
!-------------------------------------------------------------------------------
PROGRAM LIBRARY_USERS
   USE CLASS_PERSON_POINTER
   IMPLICIT NONE
   INTEGER I
   INTEGER N_USERS, N_PR, N_ST, N_PE
   TYPE(PERSON_POINTER),ALLOCATABLE :: USER(:)
   TYPE(PERSON)   ,TARGET,ALLOCATABLE :: PE(:)
```

```
   TYPE(STUDENT)  ,TARGET,ALLOCATABLE :: ST(:)
   TYPE(PROFESSOR),TARGET,ALLOCATABLE :: PR(:)
   ! N_USER = INQUIRE_NUMBER_OF_USERS() ! OBTAINS NUMBER OF USERS (NOT IMPLEMENTED)
   N_PE = 1; N_ST = 2; N_PR =1; N_USERS = N_PE + N_ST + N_PR
   ALLOCATE(USER(N_USERS), PE(N_PE), ST(N_ST), PR(N_PR))
!-------------------------------------------------------------------------------
!  CREATES BINDING TO DIFFERENT DECLARED AND DYNAMIC DATA TYPES
!-------------------------------------------------------------------------------
   USER(1)%P => ST(1)   ! 1° USER IS A STUDENT
   USER(2)%P => PE(1)   ! 2° USER IS A COMMON PERSON
   USER(3)%P => ST(2)   ! 3° USER IS A STUDENT
   USER(4)%P => PR(1)   ! 4° USER IS A PROFESSOR
!-------------------------------------------------------------------------------
!  ACTS ON THE POLYMORPHIC DATA TYPE INDEPENDENTLY FROM DECLARED AND DYNAMIC DATA TYPES
!-------------------------------------------------------------------------------
   CALL SET_FULL_NAME(USER(1)%P,'GREGORIO ','DENER ','DONATO')
   CALL SET_FULL_NAME(USER(2)%P,'JOSE',' ','MONTEIRO')
   CALL SET_FULL_NAME(USER(3)%P,'LUI','CHENG','LIU')
   CALL SET_FULL_NAME(USER(4)%P,'ANDRE','TEOFILO','BECK')
   DO I=1,N_USERS
      CALL SET_DOB(USER(I)%P,9,4,1969+I)
      CALL UPDATE_AGE(USER(I)%P)
   ENDDO
   WRITE(*,10) ' '
   WRITE(*,10) 'LIBRARY USERS: '
   WRITE(*,10) ' '
   WRITE(*,20) 'TYPE','NAME                  ','ID ','AGE'
   DO I=1,N_USERS
      WRITE(*,100) GET_PERSON_TYPE(USER(I)%P),GET_FULL_NAME(USER(I)%P), &
                   GET_ID(USER(I)%P),GET_AGE(USER(I)%P)
   ENDDO
   PAUSE
!-------------------------------------------------------------------------------
   DEALLOCATE(USER, ST, PE, PR)
10 FORMAT(A)
20 FORMAT ('  ',A12,' ',A22,' ',A3,' ',A5)
100 FORMAT('  ',A12,' ',A22,' ',I3,' ',I5)
END PROGRAM LIBRARY_USERS
```

Figure 8. Program illustrating use of polymorphic class PERSON in FORTRAN 2003.

7. POLYMORPHISM BY EMULATION IN FORTRAN 90/95

Resources of inheritance and polymorphism, obtained in Intel compiler version 11.1 through the EXTENDS keyword (*type extension*), neither are available in older compilers nor are a part of FORTRAN 90/95 standards. Hence, there are no resources for object oriented programming in these standards or in older compilers. Even so, it is possible to emulate (i.e., to mimic) polymorphism in such compilers, following (Akin, 2003; Decyk *et al*., 1997, 1998).

Examples presented in Section 6 are constructed again in this section, but using FORTRAN 90/95 standard and Compaq Visual Fortran compiler version 6.6.0. Taking into account the availability of polymorphism resources in the new Intel compiler, there are two reasons to illustrate emulation of these resources in older compilers:

a. to address users which have no access to the new Intel compiler and;
b. to show general FORTRAN users, in a very explicit form, how the new resources are implemented, automatically, in the new compiler.

This illustration will likely be elucidative for users familiar with FORTRAN 90/05 standard. However, it will be apparent that the emulation of polymorphism in older compilers requires a significantly larger number of code lines.

In the absence of type extension and inheritance, the STUDENT *derived data type* has to be created exclusively by composition, as illustrated in Fig. 9. In this case, the STUDENT type does not inherit parent components; these become sub-components if the *derived data type*:

```
STUDENT%PERSON%NAME
STUDENT%PERSON%MIDDLENAME
STUDENT%PERSON%SURNAME
```

The syntax is not so natural as that obtained with the *type extension* resource. The example in Fig. 9 illustrates only one of the many functions of the type STUDENT. Function GET_STUDENT_FULL_NAME returns a string with the full name of the student, by using the function GET_PERSON_FULL_NAME (illustrated in Fig. 5). It is noted that the function GET_STUDENT_FULL_NAME only acts as a mask to call the original function GET_PERSON_FULL_NAME with the correct parameter.

```
!-------------------------------------------------------------------------------
! IMPLEMENTATION OF CLASS STUDENT IN FORTRAN 90/95 (COMPAQ VISUAL FORTRAN V. 6.6.0).
! BY COMPOSITION, TYPE STUDENT USES COMPONENTS AND METHODS OF CLASSES DATE AND PERSON
!-------------------------------------------------------------------------------
MODULE CLASS_STUDENT
   USE CLASS_PERSON
   IMPLICIT NONE
   TYPE STUDENT
      PRIVATE
      TYPE(PERSON)        :: PERSON
```

```
      INTEGER             :: ID = 1     ! NUMBER OF SCHOLAR IDENTITY
      TYPE(CALENDAR_DATE) :: ENROLMENT  ! DATE OF SCHOOL REGISTRATION
      TYPE(CALENDAR_DATE) :: CONCLUSION ! DATE OF COMPLETION OF PROGRAM
   END TYPE STUDENT
!-------------------------------------------------------------------------------
   CONTAINS
!-------------------------------------------------------------------------------
! C = GET_STUDENT_FULL_NAME(ST) ! RETURNS FULL NAME
!-------------------------------------------------------------------------------
   FUNCTION GET_STUDENT_FULL_NAME(ST) RESULT(FULL_NAME)
      TYPE(STUDENT) ST
      CHARACTER(3*NS) FULL_NAME
      FULL_NAME(1:3*NS) = GET_PERSON_FULL_NAME(ST%PERSON)
   END FUNCTION GET_STUDENT_FULL_NAME
END MODULE CLASS_STUDENT
```

Figure 9. Implementation of class STUDENT by composition using FORTRAN 90/95.

Polymorphic *data type* PERSON (in fact, PERSON_POINTER) is created, in FORTRAN 90/95, with pointer components, as illustrated in Fig. 10. Data type PERSON_POINTER has one component which can *point* to each one of the derived data types. In run-time, the association (dynamic binding) is done with respect to one of these *data types*, and undone with respect to all the others. Function ASSOCIATE_TO_PERSON(PE), for example, associates polymorphic data type PERSON_POINTER to a data type PERSON. The same function eliminates any eventual association to other data types through the statement NULLIFY().

In addition to the association functions, it is necessary to create a mask for each one of the functions of the parent class. Figure 10 shows function GET_FULL_NAME(PT), which, according to the associated dynamic data type, calls the proper function with the correct parameters. It is noted that this function plays the role of the statement SELECT TYPE in FORTRAN 2003 (Fig. 7).

```
!-------------------------------------------------------------------------------
! IMPLEMENTATION OF "POLYMORPHIC" CLASS PERSON_POINTER IN FORTRAN 90/95 (COMPAQ VISUAL FORTRAN VERSION 6.6.0)
!-------------------------------------------------------------------------------
MODULE CLASS_PERSON_POINTER
   USE CLASS_PERSON
   USE CLASS_STUDENT
   USE CLASS_PROFESSOR
   IMPLICIT NONE
   TYPE PERSON_POINTER
      PRIVATE
      TYPE(PERSON   ),POINTER :: PERSON    ! POINTER TO A PERSON TYPE
      TYPE(STUDENT  ),POINTER :: STUDENT   ! POINTER TO A STUDENT TYPE
      TYPE(PROFESSOR),POINTER :: PROFESSOR ! POINTER TO A PROFESSOR TYPE
   END TYPE PERSON_POINTER
!-------------------------------------------------------------------------------
   CONTAINS
!-------------------------------------------------------------------------------
!  PT = ASSOCIATE_TO_PERSON(PE) ! ASSOCIATES POLYMORPHIC TYPE TO A PERSON TYPE
!-------------------------------------------------------------------------------
   FUNCTION ASSOCIATE_TO_PERSON(PE) RESULT(PT)
      TYPE(PERSON),TARGET,INTENT(IN) :: PE
      TYPE(PERSON_POINTER)           :: PT
      PT%PERSON => PE                ! ASSOCIATION OF POINTER TO PERSON TYPE
      NULLIFY(PT%STUDENT)
      NULLIFY(PT%PROFESSOR)
   END FUNCTION ASSOCIATE_TO_PERSON
!-------------------------------------------------------------------------------
                         ! ASSOCIATION FUNCTIONS FOR STUDENT AND PROFESSOR FOLLOW
!-------------------------------------------------------------------------------
! C = GET_FULL_NAME(PT)  ! RETURNS A STRING WITH THE FULL NAME
!-------------------------------------------------------------------------------
   FUNCTION GET_FULL_NAME(PT) RESULT(FULL_NAME)
      TYPE(PERSON_POINTER) :: PT
      CHARACTER(3*NS)         FULL_NAME
      IF(ASSOCIATED(PT%PERSON))    FULL_NAME(1:3*NS)=GET_PERSON_FULL_NAME(PT%PERSON)
      IF(ASSOCIATED(PT%STUDENT))   FULL_NAME(1:3*NS)=GET_STUDENT_FULL_NAME(PT%STUDENT)
      IF(ASSOCIATED(PT%PROFESSOR))FULL_NAME(1:3*NS)=GET_PROFESSOR_FULL_NAME(PT%PROFESSOR)
   END FUNCTION GET_FULL_NAME
END MODULE CLASS_PERSON_POINTER
```

Figure 10. Implementation of "polymorphic" class PERSON_POINTER in FORTRAN 90/95.

The syntax of the program LIBRARY_USERS in FORTRAN 90/95 is very similar to that of FORTRAN 2003 version (Fig. 8). The major difference is the dynamic association in run-time, which, in this case, is done through association routines, such as:

```
USER(I) = ASSOCIATE_TO_PERSON(PE(J))
```

From this association on, the program manipulates generically `USER` objects, independently of the data type of its components. Therefore, most additional statements, necessary to "emulate" polymorphism in FORTRAN 90/95, remain hidden in the implementation of the derived classes and of the "generic" class `PERSON_POINTER`.

8. CONCLUSIONS

In this article, some concepts of oriented object programming were presented: data abstraction, classes, objects, encapsulation and data access restriction, polymorphism, and construction of classes by composition and inheritance. The implementation of these concepts using FORTRAN 90/95/2003 was also presented. It was shown in the paper that object-oriented programming resources of FORTRAN 2003 standard are available in Intel Visual FORTRAN compiler version 11.1, released recently. With these resources, it is already possible to create polymorphic classes and to develop FORTRAN programs following the object orientation paradigm.

The concept of *abstract data types* is, by definition, independent from programming language. In FORTRAN, the so-called *user-defined data types* make it possible to implement ADTs in construction of classes. FORTRAN modules allow encapsulation of class contents, thus protecting information. Specific resources for object oriented programming, such as inheritance and polymorphism, were introduced by FORTRAN 2003 standard and are available in Intel Visual FORTRAN compiler version 11.1.

The article also showed that, by the use of pointers and by composition, it is possible to mimic the behavior of polymorphic classes in FORTRAN 90/95, even though it results in additional lines of code. Emulation of polymorphism through pointers gives an explicit idea about how the automatic polymorphism was implemented in the new Intel compiler.

It is important to note that the concepts presented in this paper are perfectly applicable in practice and were already incorporated into a number of programs developed by the first author and collaborators. This includes a program for structural reliability analysis, having thirteen thousand lines of code, developed in FORTRAN 90/95 and already extensively tested. The two versions of the university management program, used to illustrate OOP concepts throughout this article, are incomplete but are perfectly functional. The FORTRAN 90/95 version of this program performs the management of university classes and already reached two thousand three hundred lines of code.

The experience of the first author shows that, with the resources available in Intel Visual FORTRAN compiler version 11.1, and even with the resources existing in FORTRAN 90/95, it is possible to reach the main objectives of object oriented programming, that is: expansibility, reusability, and compatibility of programs developed in FORTRAN. The object oriented programming paradigm allows us to develop large computer programs in a clear, logical and consistent form.

9. ACKNOWLEDGEMENTS

Sponsorship of this research project by the Brazilian National Council for Research and Development (CNPq), the Brazilian National Council for Higher Degree Education (CAPES) and the São Paulo Research Foundation (FAPESP) is greatly acknowledged.

10. REFERENCES

Adams, J.C., Brainerd, W.S., Martin, J.T., Smith, B.T. and Wagener, J.L., 1992, "Fortran 90 Handbook: Complete ANSI/ISO Reference", Intertext Publications, McGraw-Hill, New York, USA.

Adams, J.C., Brainerd, W.S., Martin, J.T., Smith, B.T. and Wagener, J.L., 1997, "Fortran 95 Handbook: Complete ISO/ANSI Reference", MIT Press, USA.

Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T. and Smith, B.T., 2009, "The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures", Springer, London, England.

Akin, E., 2003, "Object Oriented Programming via FORTRAN 90/95", Cambridge University Press.

Chapman, S.J., 2007, "Fortran 95/2003 for Scientists and Engineers", 3rd Edition, McGraw-Hill, USA.

Decyk, V.K., Norton, C.D., Szymanski, B.K., 1997, "Expressing object-oriented concepts in Fortran 90", ACM Fortran Forum, Vol. 16, No. 1 (April), New York, NY, USA, pp. 13-18.

Decyk, V.K., Norton, C.D., Szymanski, B.K., 1998. "How to support inheritance and run-time polymorphism in Fortran 90". Computer Physics Communications, Vol. 115, No. 1, pp. 9-17.

Intel, 2009, "Intel® Visual Fortran Compiler Professional Edition 11.1 for Windows*, Instalation Guide and Release Notes", October.

Meyer, B., 2000, "Object-Oriented Software Construction", 2nd edition, Prentice-Hall.

11. RESPONSIBILITY NOTICE

The authors are the only responsible for the printed material included in this paper.