Object Oriented Programming via Fortran 90

(Preprint: Engineering Computations, v. 16, n. 1, pp. 26-48, 1999)

J. E. Akin
Rice University, MEMS Dept.
Houston, TX 77005-1892

Keywords object-oriented, encapsulation, inheritance, polymorphism, Fortran 90

Abstract

There is a widely available object-oriented (OO) programming language that is usually overlooked in the OO Analysis, OO Design, OO Programming literature. It was designed with most of the features of languages like C++, Eiffel, and Smalltalk. It has extensive and efficient numerical abilities including concise array and matrix handling, like Matlab®. In addition, it is readily extended to massively parallel machines and is backed by an international ISO and ANSI standard. The language is Fortran 90 (and Fortran 95).

When the explosion of books and articles on OOP began appearing in the early 1990's many of them correctly disparaged Fortran 77 (F77) for its lack of object oriented abilities and data structures. However, then and now many authors fail to realize that the then new Fortran 90 (F90) standard established a well planned object oriented programming language while maintaining a full backward compatibility with the old F77 standard. F90 offers strong typing, encapsulation, inheritance, multiple inheritance, polymorphism, and other features important to object oriented programming. This paper will illustrate several of these features that are important to engineering computation using OOP.

1. Introduction

The use of Object Oriented (OO) design and Object Oriented Programming (OOP) is becoming increasingly popular (Coad, 1991; Filho, 1991; Rumbaugh, 1991), and today there are more than 100 OO languages. Thus, it is useful to have an introductory understanding of OOP and some of the programming features of OO languages. You can develop OO software in any high level language, like C or Pascal. However, newer languages such as Ada, C++, and F90 have enhanced features that make OOP much more natural, practical, and maintainable. C++ appeared before F90 and currently, is probably the most popular OOP language, yet F90 was clearly designed to have almost all of the abilities of C++ (Adams, 1992; Barton, 1994). However, rather than study the new standards many authors simply refer to the two decades old F77 standard and declare that Fortran can not be used for OOP. Here we will try to overcome that misinformed point of view.

Modern OO languages provide the programmer with three capabilities that improve and simplify the design of such programs: encapsulation, inheritance, and polymorphism (or generic functionality). Related topics involve objects, classes, and data hiding. An object combines various classical data types into a set that defines a new variable type, or structure. A class unifies the new entity types and supporting data that represents its status with subprograms (functions and subroutines) that access and/or modify those data. Every object created from a class, by providing the necessary data, is called an instance of the class. In older languages like C and F77, the data and functions are separate entities. An OO language provides a way to couple or encapsulate the data and its functions into a unified entity. This is a more natural way to model real-world entities which have both data and functionality. The encapsulation is done with a "module" block in F90, and with a "class" block in C++. This encapsulation also includes a mechanism whereby some or all of the data and supporting subprograms can be hidden from the user. The accessibility of the specifications and subprograms of a class is usually controlled by optional "public" and "private" qualifiers. Data hiding allows one the means to protect information in one part of a program from access, and especially from being changed in other parts of the program. In C++ the default is that data and functions are "private" unless declared "public," while F90 makes the opposite choice for its default protection mode. In a F90 "module" it is the "contains" statement that, among other things, couples the data, specifications, and operators before it to the functions and subroutines that follow it.

Class hierarchies can be visualized when we realize that we can employ one or more previously defined classes (of data and functionality) to organize additional classes. Functionality programmed into the earlier classes may not need to be re-coded to be usable in the later classes. This mechanism is called inheritance. For example, if we have defined an **Employee_class**, then a **Manager_class** would inherit all of the data and functionality of an employee. We would then only be required to add only the totally new data and functions needed for a manager. We may also need a mechanism to re-define specific **Employee_class** functions that differ for a **Manager_class**. By using the concept of a class hierarchy, less programming effort is required to create the final enhanced program. In F90 the earlier class is brought into the later class hierarchy by the **use** statement followed by the name of the "module" statement block that defined the class.

Polymorphism allows different classes of objects that share some common functionality to be used in code that requires only that common functionality. In other words, subprograms having the same generic name are interpreted differently depending on the class of the objects presented as arguments to the subprograms. This is useful in class hierarchies where a small number of meaningful function names can be used to manipulate different, but related object classes. The above concepts are those essential to object oriented design and OOP. In the later sections we will demonstrate by example F90 implementations of these concepts.


```
 !   Areas of shapes of different classes, using different
 !            function names in each class
 module class_Rectangle    ! define the first object class
   type Rectangle
     real :: base, height ; end  type Rectangle

 contains !  Computation of area for rectangles.
   function rectangle_area ( r ) result ( area )
     type ( Rectangle ), intent(in) :: r
     real                           :: area
       area = r%base * r%height ; end function rectangle_area
 end module class_Rectangle

 module class_Circle    ! define the second object class
   real :: pi = 3.1415926535897931d0 ! a circle constant
   type Circle
     real :: radius ; end  type Circle

 contains !      Computation of area for circles.
   function circle_area ( c ) result ( area )
     type ( Circle ), intent(in) :: c
     real                        :: area
       area = pi * c%radius**2 ; end function circle_area
 end module class_Circle

program geometry  ! for both types in a single function
 use class_Circle
 use class_Rectangle

!   Interface to generic routine to compute area for any type
  interface compute_area
    module procedure rectangle_area, circle_area ; end interface

 !     Declare a set geometric objects.
  type ( Rectangle ) :: four_sides
  type ( Circle    ) :: two_sides        ! inside, outside
  real               :: area = 0.0       ! the result

 !     Initialize a rectangle and compute its area.
    four_sides = Rectangle ( 2.1, 4.3 )    ! implicit constructor
    area = compute_area ( four_sides )     ! generic function
    write ( 6,100 ) four_sides, area       ! implicit components list
    100 format ("Area of ",f3.1," by ",f3.1," rectangle is ",f5.2)

 !     Initialize a circle and compute its area.
    two_sides = Circle ( 5.4 )             ! implicit constructor
    area = compute_area ( two_sides )      ! generic function
    write ( 6,200 ) two_sides, area
    200 format ("Area of circle with ",f3.1," radius is ",f9.5 )
 end program geometry                   ! Running gives:
! Area of 2.1 by 4.3 rectangle is  9.03
! Area of circle with 5.4 radius is  91.60885
```

Figure 1: *Multiple Geometric Shape Classes*

2. Encapsulation, Inheritance, and Polymorphism

We often need to use existing classes to define new classes. The two ways to do this are called composition and inheritance. We will use both methods in a series of examples. Consider a geometry program that uses two different classes: **class_Circle** and **class_Rectangle**, such as that shown in Figure 1 on page 3. Each class shown has the data types and specifications to define the object and the functionality to compute their respective areas. The operator **%** is employed to select specific components of a defined type. Within the geometry (main) program a single subprogram, **compute_area**, is invoked to return the area for any of the defined geometry classes. That is, a generic function name is used for all classes of its arguments and it, in turn, branches to the corresponding functionality supplied with the argument class. To accomplish this branching the geometry program first brings in the functionality of the desired classes via a **use** statement for each class module. Those "modules" are coupled to the generic function by an **interface** block which has the generic function name (**compute_area**). There is included a **module procedure** list which gives one class subprogram name for each of the classes of argument(s) that the generic function is designed to accept. The ability of a function to respond differently when supplied with arguments that are objects of different types is called polymorphism. In this example we have employed different names, **rectangular_area** and **circle_area**, in their respective class modules, but that is not necessary. The **use** statement allows one to rename the class subprograms and/or to bring in only selected members of the functionality.

Another terminology used in OOP is that of constructors and destructors for objects. An intrinsic constructor is a system function that is automatically invoked when an object is declared with all of its possible components in the defined order. In C++, and F90 the intrinsic constructor has the same name as the "type" of the object. One is illustrated in Figure 1 on page 3 in the statement:

```
four_sides = Rectangle (2.1,4.3)
```

where previously we declared

```
type (Rectangle) :: four_sides
```

which, in turn, was coupled to the **class_Rectangle** which had two components, base and height, defined in that order, respectively. The intrinsic constructor in the example statement sets component base = 2.1 and component height = 4.3 for that instance, **four_sides**, of the type **Rectangle**. This intrinsic construction is possible because all the expected components of the type were supplied. If all the components are not supplied, then the object cannot be constructed unless the functionality of the class is expanded by the programmer to accept a different number of arguments.

Assume that we want a special member of the **Rectangle** class, a square, to be constructed if the height is omitted. That is, we would use height = base in that case. Or, we may want to construct a unit square if both are omitted so that the constructor defaults to base = height = 1. Such a manual constructor, named **make_Rectangle**, is illustrated in Figure 2 on page 5. It illustrates some additional features of F90. Note that the last two arguments were declared to have the additional type attributes of **optional**, and that an associated logical function **present** is utilized to determine if the calling program supplied the argument in question. That figure also shows the results of the area computations for the corresponding variables **square** and **unit_sq** defined if the manual constructor is called with one or no optional arguments, respectively.

```
function  make_Rectangle (bottom, side) result (name)
!          Constructor for a Rectangle type
  real, optional, intent(in) :: bottom, side
  type (Rectangle)           :: name
    name = Rectangle (1.,1.)      ! default to unit square
    if ( present(bottom) ) then   ! default to square
      name = Rectangle (bottom, bottom) ; end if
    if ( present(side) ) name = Rectangle (bottom, side) ! intrinsic
end function  make_Rectangle
 . . .
 type ( Rectangle ) :: four_sides, square, unit_sq

!     Test manual constructors
   four_sides = make_Rectangle (2.1,4.3)   ! manual constructor, 1
   area = compute_area ( four_sides)       ! generic function
   write ( 6,100 ) four_sides, area

!     Make a square
   square = make_Rectangle (2.1)           ! manual constructor, 2
   area = compute_area ( square)           ! generic function
   write ( 6,100 ) square, area

!     "Default constructor", here a unit square
   unit_sq = make_Rectangle ()             ! manual constructor, 3
   area = compute_area (unit_sq)           ! generic function
   write ( 6,100 ) unit_sq, area  ! Running gives:
! Area of 2.1 by 4.3 rectangle is  9.03
! Area of 2.1 by 2.1 rectangle is  4.41
! Area of 1.0 by 1.0 rectangle is  1.00
```

Figure 2: *A Manual Constructor for Rectangles*

Before moving to some mathematical examples we will introduce the concept of data hiding and combine a series of classes to illustrate composition and inheritancey. First, consider a simple class to define dates and to print them in a pretty fashion. While other modules will have access to the Date class they will not be given access to the number of components it contains (3), nor their names (month, day, year), nor their types (integers) because they are declared **private** in the defining module. The compiler will not allow external access to data and/or subprograms declared as private. The module, **class_Date**, is presented as a source **include** file in Figure 3 on page 6, and in the future will be reference by the file name **class_Date.f90**. Since we have chosen to hide all the user defined components we must decide what functionality we will provide to the users, who may have only executable access. The supporting documentation would have to name the public subprograms and describe their arguments and return results. The default intrinsic constructor would be available only to those that know full details about the components of the data type, and if those components are **public**. The intrinsic constructor, **Date**, requires all the components be supplied, but it does no error or consistency checks. My practice is to also define a "public constructor" whose name is the same as the intrinsic constructor except for an appended underscore, that is, **Date_**. Its sole purpose is to do data checking and invoke the intrinsic constructor, **Date**. If the function **Date_** is declared **public** it can be used outside the module **class_Date** to invoke the intrinsic constructor, even if the components of the data type being constructed are all **private**. In this example we have provided another manual constructor to set a date, **set_Date**, with a variable number of optional arguments. Also supplied are two subroutines to read and print dates, **read_Date** and **print_Date**, respectively.

```
module class_Date        ! filename: class_Date.f90
  public  :: Date ! and everything not "private"   type Date
    private
    integer :: month, day, year ; end type Date

contains ! encapsulated functionality

function Date_ (m, d, y) result (x) ! public constructor
  integer, intent(in) :: m, d, y    ! month, day, year
  type (Date)         :: x          ! from intrinsic constructor
    if ( m < 1 .or. d < 1 ) stop 'Invalid components, Date_'
    x = Date (m, d, y) ; end function Date_

subroutine  print_Date (x)    ! check and pretty print a date
  type (Date),      intent(in) :: x
  character (len=*),parameter  :: month_Name(12) = &
   (/ "January  ", "February ", "March    ", "April    ",&
      "May      ", "June     ", "July     ", "August   ",&
      "September", "October  ", "November ", "December "/)
    if ( x%month < 1 .or. x%month > 12 ) print *, "Invalid month"
    if ( x%day   < 1 .or. x%day   > 31 ) print *, "Invalid day  "
    print *, trim(month_Name(x%month)),' ', x%day, ", ", x%year;
 end subroutine  print_Date

subroutine  read_Date (x)        ! read month, day, and year
  type (Date), intent(out) :: x  ! into intrinsic constructor
    read *, x ; end subroutine  read_Date

function set_Date (m, d, y) result (x)     ! manual constructor
  integer, optional, intent(in) :: m, d, y ! month, day, year
  type (Date)                   :: x
    x = Date (1,1,1997)         ! default, (or use current date)
    if ( present(m) ) x%month = m ; if ( present(d) ) x%day = d
    if ( present(y) ) x%year  = y ; end function set_Date

end module class_Date
```

Figure 3: *Defining a Date Class*

A sample main program that employs this class is given in Figure 4 on page 7, which contains sample outputs as comments. This program uses the default constructor as well as all three programs in the public class functionality. Note that the definition of the class was copied in via an include statement and activated with the **use** statement.

```
include 'class_Date.f90'   ! see previous figure

program  main
 use class_Date
  type (Date) :: today, peace

  ! peace = Date (11,11,1918) ! NOT allowed for private components
    peace = Date_ (11,11,1918)               ! public constructor
    print *, "World War I ended on "  ; call print_Date (peace)
    peace = set_Date (8, 14, 1945)           ! optional constructor

    print *, "World War II ended on " ; call print_Date (peace)
    print *, "Enter today as integer month, day, and year: "
    call read_Date(today)                   ! create today's date
    print *, "The date is ";  call print_Date (today)
end program  main                         ! Running produces:
! World War I ended on November 11, 1918
! World War II ended on August 14, 1945
! Enter today as integer month, day, and year: 7 10 1998
! The date is July 10, 1998
```

Figure 4: *Testing a Date Class*

Now we will employ the **class_Date** within a **class_Person** which will use it to set the date of birth (DOB) and date of death (DOD) in addition to the other **Person** components of name, nationality, and sex. Again we have made all the type components **private**, but make all the supporting functionality **public**. The functionality shown provides a manual constructor, **make_Person**, subprograms to set the DOB or DOD, and those for the printing of most components. The new class is given in Figure 5 on page 8. Note that the manual constructor utilizes **optional** arguments and initializes all components in case they are not supplied to the constructor. The **set_Date** public subroutine from the **class_Date** is "inherited" to initialize the DOB and DOD. That function member from the previous module was activated with the combination of the **include** and **use** statements. Of course, the **include** could have been omitted if the compile statement included the path name to that source. A sample main program for testing the **class_Person** is in Figure 6 on page 9 along with comments containing its output.

```
 module class_Person          ! filename: class_Person.f90
 use class_Date
  public  :: Person
    type Person
      private
      character (len=20) :: name
      character (len=20) :: nationality
      integer            :: sex
      type (Date)        :: dob, dod    ! birth, death
```



```
    end type Person
 contains

 function make_Person (nam, nation, s, b, d) result (who)
  !        Optional Constructor for a Person type
   character (len=*), optional, intent(in) :: nam, nation
   integer,           optional, intent(in) :: s     ! sex
   type (Date),       optional, intent(in) :: b, d  ! birth, death
   type (Person)                           :: who
     who = Person (" ","USA",1,Date_(1,1,0),Date_(1,1,0))! defaults
     if ( present(nam)    ) who % name        = nam
     if ( present(nation) ) who % nationality = nation
     if ( present(s)      ) who % sex         = s
     if ( present(b)      ) who % dob         = b
     if ( present(d)      ) who % dod         = d ; end function

function Person_ (nam, nation, s, b, d) result (who)
 !        Public Constructor for a Person type
  character (len=*), intent(in) :: nam, nation
  integer,           intent(in) :: s     ! sex
  type (Date),       intent(in) :: b, d  ! birth, death
  type (Person)                 :: who
    who = Person (nam, nation, s, b, d) ; end function Person_

subroutine print_DOB (who)
  type (Person), intent(in) :: who
    call  print_Date (who % dob) ; end subroutine  print_DOB

subroutine print_DOD (who)
  type (Person), intent(in) :: who
    call print_Date (who % dod) ; end subroutine  print_DOD

subroutine print_Name (who)
  type (Person), intent(in) :: who
    print *, who % name ; end subroutine print_Name

subroutine print_Nationality (who)
  type (Person), intent(in) :: who
    print *, who % nationality ; end subroutine print_Nationality

subroutine print_Sex (who)
  type (Person), intent(in) :: who
    if ( who % sex == 1 ) then ; print *, "male"
    else ; print *, "female" ; end if ; end subroutine print_Sex

subroutine set_DOB (who, m, d, y)
  type (Person), intent(inout) :: who
  integer, intent(in)          :: m, d, y ! month, day, year
    who % dob = Date_ (m, d, y) ;  end subroutine set_DOB

subroutine set_DOD(who, m, d, y)
  type (Person), intent(inout) :: who
  integer, intent(in)          :: m, d, y ! month, day, year
    who % dod = Date_ (m, d, y) ;  end subroutine set_DOD
end module class _Person
```

Figure 5: *Definition of a Typical Person Class*


```
include 'class_Date.f90'
include 'class_Person.f90'                   ! see previous figure
program main
 use class_Date ; use class_Person           ! inherit class members
   type (Person) :: author, creator
   type (Date)   :: b, d                      ! birth, death
   b = Date_(4,13,1743) ; d = Date_(7, 4,1826) ! OPTIONAL

!                      Method 1
!  author = Person ("Thomas Jefferson", "USA", 1, b, d)  ! iff private
   author = Person_ ("Thomas Jefferson", "USA", 1, b, d) ! constructor
   print *,"The author of the Declaration of Independence was ";
   call  print_Name (author);
   print *,". He was born on "; call print_DOB (author);
   print *," and died on ";     call print_DOD (author); print *,".";

 !                     Method 2
   author = make_Person ("Thomas Jefferson", "USA") ! alternate
   call  set_DOB (author, 4, 13, 1743)              ! add DOB
   call  set_DOD (author, 7,  4, 1826)              ! add DOD
   print *,"The author of the Declaration of Independence was ";
   call  print_Name (author)
   print *,". He was born on "; call print_DOB (author);
   print *," and died on ";     call print_DOD (author); print *,".";

 !                     Another Person
   creator = make_Person ("John Backus", "USA")     ! alternate
   print *,"The creator of Fortran was "; call print_Name (creator);
   print *," who was born in ";    call print_Nationality (creator);
   print *,".";
 end program main                                   ! Running gives:
! The author of the Declaration of Independence was Thomas Jefferson.
! He was born on April 13, 1743 and died on July 4, 1826.
! The author of the Declaration of Independence was Thomas Jefferson.
! He was born on April 13, 1743 and died on July 4, 1826.
! The creator of Fortran was John Backus who was born in the USA.
```

Figure 6: *Testing the Date and Person Classes*

Next, we want to use the previous two classes to define a **class_Student** which adds something else special to the general **class_Person**. The **Student** person will have additional **private** components for an identification number, the expected date of matriculation (DOM), the total course credit hours earned (credits), and the overall grade point average (GPA). The type definition and selected public functionality are given if Figure 7 on page 10 while a testing main program with sample output is illustrated in Figure 8 on page 11. Since there are various ways to utilize the various constructors some alternate source lines have been included as comments to indicate some of the programmer's options.


```
module class_Student           ! filename class_Student.f90
 use class_Person               ! inherits class_Date
  public :: Student, set_DOM, print_DOM
    type Student
      private
      type (Person)     :: who    ! name and sex
      character (len=9) :: id     ! ssn digits
      type (Date)       :: dom    ! matriculation
      integer           :: credits
      real              :: gpa    ! grade point average
    end type Student
 contains  ! coupled functionality
       function get_person (s) result (p)
         type (Student), intent(in) :: s
         type (Person)              :: p      ! name and sex
           p = s % who ; end function get_person

       function make_Student (w, n, d, c, g) result (x)
      !        Optional Constructor for a Student type
        type (Person),                intent(in) :: w ! who
        character (len=*), optional, intent(in) :: n ! ssn
        type (Date),       optional, intent(in) :: d ! matriculation
        integer,           optional, intent(in) :: c ! credits
        real,              optional, intent(in) :: g ! grade point ave
        type (Student)                          :: x ! new student
          x = Student_(w, " ", Date_(1,1,1), 0, 0.)  ! defaults
          if ( present(n) ) x % id      = n          ! optional values
          if ( present(d) ) x % dom     = d
          if ( present(c) ) x % credits = c
          if ( present(g) ) x % gpa     = g; end function make_Student

       subroutine print_DOM (who)
         type (Student), intent(in) :: who
           call print_Date(who%dom) ; end subroutine print_DOM

       subroutine print_GPA (x)
         type (Student), intent(in) :: x
           print *,"My name is "; call print_Name (x % who)
           print *,", and my G.P.A. is ", x % gpa, "."; end subroutine

       subroutine set_DOM (who, m, d, y)
         type (Student), intent(inout) :: who
           integer,        intent(in)    :: m, d, y
           who % dom = Date_( m, d, y) ; end subroutine set_DOM

       function Student_ (w, n, d, c, g) result (x)
       !        Public Constructor for a Student type
         type (Person),     intent(in) :: w ! who
         character (len=*), intent(in) :: n ! ssn
         type (Date),       intent(in) :: d ! matriculation
         integer,           intent(in) :: c ! credits
         real,              intent(in) :: g ! grade point ave
         type (Student)                :: x ! new student
           x = Student (w, n, d, c, g) ; end function Student_
 end module class_Student
```

Figure 7: *Defining a Typical Student Class*

```
include 'class_Date.f90'
include 'class_Person.f90'
include 'class_Student.f90' ! see previous figure
program main                ! create or correct a student
 use class_Student          ! inherits class_Person, class_Date also
  type (Person) :: p  ; type (Student) :: x

 !              Method 1
    p = make_Person ("Ann Jones","",0)  ! optional person constructor
    call set_DOB (p, 5, 13, 1977)       ! add birth to person data
    x = Student_(p, "219360061", Date_(8,29,1955), 9, 3.1) ! public
    call print_Name (p)                               ! list name
    print *, "Born        :"; call print_DOB (p)     ! list dob
    print *, "Sex         :"; call print_Sex (p)     ! list sex
    print *, "Matriculated:"; call print_DOM (x)     ! list dom
    call print_GPA (x)                               ! list gpa

 !              Method 2
    x = make_Student (p, "219360061")  ! optional student constructor
    call set_DOM (x, 8, 29, 1995)      ! correct matriculation
    call print_Name (p)                               ! list name
    print *, "was born on :"; call print_DOB (p)     ! list dob
    print *, "Matriculated:"; call print_DOM (x)     ! list dom

 !              Method 3
    x = make_Student (make_Person("Ann Jones"),"219360061")! optional
    p = get_Person (x)                  ! get defaulted person data
    call set_DOM (x, 8, 29, 1995)       ! add matriculation
    call set_DOB (p, 5, 13, 1977)       ! add birth
    call print_Name (p)                               ! list name
    print *, "Matriculated:"; call print_DOM (x)     ! list dom
    print *, "was born on :"; call print_DOB (p)     ! list dob
end program main                                   ! Running gives:
! Ann Jones
! Born        : May 13, 1977
! Sex         : female
! Matriculated: August 29, 1955
! My name is Ann Jones, and my G.P.A. is 3.0999999.
! Ann Jones was born on: May 13, 1977, Matriculated: August 29, 1995
! Ann Jones Matriculated: August 29, 1995, was born on: May 13, 1977
```

Figure 8: *Testing the Student, Person, and Date Classes*

3. Object Oriented Numerical Calculations

OOP is often used for numerical computation, especially when the standard storage mode for arrays is not practical or efficient. Often one will find specialized storage modes like linked lists (Akin, 1997; Barton, 1994; Hubbard, 1994), or tree structures used for dynamic data structures. Here we should note that many matrix operators are intrinsic to F90, so one is more likely to define a **class_sparse_matrix** than a **class_matrix**. However, either class would allow us to encapsulate several matrix functions and subroutines into a module that could be reused easily in other software. Here, we will illustrate OOP applied to rational numbers and vectors and introduce the important topic of operator overloading.

3.1 A Rational Number Class and Operator Overloading

To illustrate an OOP approach to simple numerical operations we will introduce a fairly complete rational number class, called **class_Rational**. The defining module is given in Figure 9 on page 14. The type components have been made private, but not the type itself, so we can illustrate the intrinsic constructor, but extra functionality has been provided to allow users to get either of the two components. The provided subprograms shown in that figure are:

```
add_Rational   convert      copy_Rational    delete_Rational
equal_integer  gcd          get_Denominator  get_Numerator
invert         is_equal_to  list             make_Rational
mult_Rational  Rational     reduce
```

Procedures with only one return argument are usually implemented as functions instead of subroutines.

Note that we would form a new rational number, z, as the product of two other rational numbers, x and y, by invoking the **mult_Rational** function**,**

```
z = mult_Rational (x, y)
```

which returns z as its result. A natural tendency at this point would be to simply write this as z = x * y. However, before we could do that we would have to have to tell the operator, "*", how to act when provided with this new data type. This is known as overloading an intrinsic operator. We had the foresight to do this when we set up the module by declaring which of the "module procedures" were equivalent to this operator symbol. Thus, from the **interface operator (*)** statement block the system now knows that the left and right operands of the "*" symbol correspond to the first and second arguments in the function **mult_Rational**. Here it is not necessary to overload the assignment operator, "=", when both of its operands are of the same intrinsic or defined type. However, to convert an integer to a rational we could, and have, defined an overloaded assignment operator procedure. Here we have provided the procedure, **equal_Integer**, which is automatically invoked when we write: **type (Rational) y; y = 4.** That would be simpler than invoking the constructor called **make_rational**.

Before moving on note that the system does not yet know how to multiply an integer times a rational number, or visa versa. To do that one would have to add more functionality, such as a function, say **int_mult_rn**, and add it to the **module procedure** list associated with the "*" operator. A typical main program which exercises most of the rational number functionality is given in Figure 10 on page 15, along with typical numerical output.


```
module class_Rational                ! filename: class_Rational.f90
 ! public, everything but following private subprograms
 private :: gcd, reduce
   type Rational
     private ! numerator and denominator
     integer :: num, den ; end type Rational

! overloaded operators interfaces
   interface assignment (=)
     module procedure equal_Integer ; end interface
   interface operator (+)          ! add unary versions & (-) later
     module procedure add_Rational ; end interface
   interface operator (*)          ! add integer_mult_Rational, etc
     module procedure mult_Rational ; end interface
   interface operator (==)
     module procedure is_equal_to ; end interface
contains                     ! inherited operational functionality
function add_Rational (a, b) result (c)       ! to overload +
  type (Rational), intent(in) :: a, b          ! left + right
  type (Rational)             :: c
    c % num = a % num*b % den + a % den*b % num
    c % den = a % den*b % den
    call reduce (c) ; end function add_Rational

function convert (name) result (value) ! rational to real
  type (Rational), intent(in) :: name
  real                        :: value  ! decimal form
    value = float(name % num)/name % den ; end function convert

function copy_Rational (name) result (new)
  type (Rational), intent(in) :: name
  type (Rational)             :: new
    new % num = name % num
    new % den = name % den ; end function copy_Rational

subroutine delete_Rational (name)     ! deallocate allocated items
  type (Rational), intent(inout) :: name     ! simply zero it here
    name = Rational (0, 1) ; end subroutine delete_Rational

subroutine equal_Integer (new, I) ! overload =, with integer
  type (Rational), intent(out) :: new  ! left  side of operator
  integer,         intent(in)  :: I    ! right side of operator
    new % num = I ; new % den = 1 ; end subroutine equal_Integer

recursive function gcd (j, k) result (g) ! Greatest Common Divisor
  integer, intent(in) :: j, k ! numerator, denominator
  integer             :: g
    if ( k == 0 ) then ; g = j
    else ; g = gcd ( k, modulo(j,k) )           ! recursive call
    end if ; end function gcd

function  get_Denominator (name) result (n) ! an access function
  type (Rational), intent(in) :: name
  integer                     :: n          ! denominator
    n = name % den ; end function  get_Denominator
```

```
function  get_Numerator (name) result (n)    ! an access function
  type (Rational), intent(in) :: name
  integer                     :: n          ! numerator
    n = name % num ; end function  get_Numerator

subroutine  invert (name)      ! rational to rational inversion
  type (Rational), intent(inout) :: name
  integer                        :: temp
    temp       = name % num
    name % num = name % den
    name % den = temp ; end subroutine invert

function is_equal_to (a_given, b_given) result (t_f)     ! for ==
  type (Rational), intent(in) :: a_given, b_given ! left == right
  type (Rational)             :: a, b             ! reduced copies
  logical                     :: t_f
    a = copy_Rational (a_given) ; b = copy_Rational (b_given)
    call reduce(a) ; call reduce(b)      ! reduced to lowest terms
    t_f = (a%num == b%num) .and. (a%den == b%den) ; end function

subroutine list(name)                 ! as a pretty print fraction
  type (Rational), intent(in) :: name
    print *, name % num, "/", name % den ; end subroutine list

function make_Rational (numerator, denominator) result (name)
!         Optional Constructor for a rational type
  integer, optional, intent(in) :: numerator, denominator
  type (Rational)               :: name
    name = Rational(0, 1)                           ! set defaults
    if ( present(numerator)  ) name % num = numerator
    if ( present(denominator)) name % den = denominator
    if ( name % den == 0     ) name % den = 1    ! now simplify
    call reduce (name) ; end function make_Rational

function  mult_Rational (a, b) result (c)        ! to overload *
  type (Rational), intent(in) :: a, b
  type (Rational)             :: c
    c % num = a % num * b % num ; c % den = a % den * b % den
    call reduce (c) ; end function mult_Rational

function Rational_ (numerator, denominator) result (name)
!         Public Constructor for a rational type
 integer, optional, intent(in) :: numerator, denominator
 type (Rational)               :: name
   if ( denominator == 0 ) then ; name = Rational (numerator, 1)
   else ; name = Rational (numerator, denominator) ; end if
end function Rational_

subroutine reduce (name) ! to simplest rational form
  type (Rational), intent(inout) :: name
  integer                        :: g  ! greatest common divisor
    g          = gcd (name % num, name % den)
    name % num = name % num/g
    name % den = name % den/g ; end subroutine reduce
end module class_Rational
```

Figure 9: *A Fairly Complete Rational Number Class*


```
!   F90 Implementation of a Rational Class Constructors & Operators
include 'class_Rational.f90'
program main
 use class_Rational
 type (Rational) :: x, y, z

! x = Rational(22,7)     ! intrinsic constructor iff public components
  x = Rational_(22,7)    ! public constructor if private components

  write (*,'("public    x  = ")',advance='no'); call list(x)
  write (*,'("converted x  = ", g9.4)') convert(x)
  call invert(x)
  write (*,'("inverted 1/x = ")',advance='no');  call list(x)
  x = make_Rational ()                ! default constructor
  write (*,'("made null x  = ")',advance='no'); call list(x)
  y = 4                              ! rational = integer overload
  write (*,'("integer y    = ")',advance='no'); call list(y)
  z = make_Rational (22,7)            ! manual constructor
  write (*,'("made full z  = ")',advance='no'); call list(z)

!                 Test Accessors
  write (*,'("top of z     = ", g4.0)') get_numerator(z)
  write (*,'("bottom of z  = ", g4.0)') get_denominator(z)

!                 Misc. Function Tests
  write (*,'("making x = 100/360, ")',advance='no')
  x = make_Rational (100,360)
  write (*,'("reduced x = ")',advance='no'); call list(x)
  write (*,'("copying x to y gives ")',advance='no')
  y = copy_Rational (x)
  write (*,'("a new y = ")',advance='no'); call list(y)

!                Test Overloaded Operators
  write (*,'("z * x gives ")',advance='no'); call list(z*x) ! times
  write (*,'("z + x gives ")',advance='no'); call list(z+x) ! add
  y = z                                    ! overloaded assignment
  write (*,'("y = z gives y as ")',advance='no'); call list(y)
  write (*,'("logic y == x gives ")',advance='no'); print *, y==x
  write (*,'("logic y == z gives ")',advance='no'); print *, y==z

!                 Destruct
  call delete_Rational (y)    ! actually only null it here
  write (*,'("deleting y gives y = ")',advance='no'); call list(y)
end program main                                     ! Running gives:
!  public    x  =  22 / 7       !  converted x  = 3.143
!  inverted 1/x =  7 / 22       !  made null x  =  0 / 1
!  integer y    =  4 / 1        !  made full z  =  22 / 7
!  top of z     =   22          !  bottom of z  =    7
!  making x = 100/360, reduced x =  5 / 18
!  copying x to y gives a new y =  5 / 18
!  z * x gives  55 / 63         !  z + x gives  431 / 126
!  y = z gives y as  22 / 7     !  logic y == x gives  F
!  logic y == z gives  T        !  deleting y gives y =  0 / 1
```

Figure 10: *Testing the Rational Number Class*

3.2 A Numerical Vector Class

Vectors are commonly used in many computational areas of engineering and applied mathematics. Thus, one might want to define a vector class that has the most commonly used operations with vectors. Of course, that is not actually required in F90 since it, like Matlab, has many intrinsic functions for operating on vectors and general arrays. However, the concepts are commonly understood, so that vectors make a good illustration of OOP for numerical applications. Also, the standard F90 features provide a simple way to verify the accuracy of our vector class procedures. Therefore, we could define a vector class, an array class that is actually a collection of vector classes, and then test them with both standard F90 features and the new OOP functionality of the two classes. The module **class_Vector** in Figure 11 on page 20 contains functions called

```
add_Real           add_Vector         assign
copy_Vector        dot_Vector         is_equal_to
length             make_Vector        normalize_Vector
real_mult_Vector   size_Vector        subtract_Real
subtract_Vector    values             vector_max_value
vector_min_value   vector_mult_real
```

and subroutines called

```
delete_Vector   equal_Real
list            read_Vector
```

where the names suggest their purpose. This OOP approach allows one to extend the available intrinsic functions and add members like **is_equal_to** and **normalize_Vector**. These subprograms are also employed to overload the standard operators (=, +, -, *, and ==) so that they work in a similar way for members of the vector class. The definitions of the vector class has also introduced the use of **pointer** variables (actually reference variables of C++) for allocating and deallocating dynamic memory for the vector coefficients as needed. Like Java, but unlike C++, F90 automatically dereferences its pointers. The availability of pointers allows the creation of storage methods like linked lists, circular lists, and trees which are more efficient than arrays for some applications (Akin, 1997). F90 also allows for the automatic allocation and deallocation of local arrays. While we have not done so here the language allows new operators to be defined to operate on members of the vector class.

The two components of the vector type are an integer that tells how many components the vector has, and then those component values are stored in a real array. Here we assume that the vectors are full and that any two vectors involved in a mathematical operation have the same number of components. Also, we do not allow the vector to have zero or negative lengths. The functionality presented here is easily extended to declare operations on a sparse vector type which is not a standard feature of F90. The first function defined in this class is **add_Real**, which will add a real number to all components in a given vector. The second function, **add_Vector**, adds the components of one vector to the corresponding components of another vector. Both were needed to overload the "+" operator so that its two operands could either be real or vector class entities. Note that the last executable statement in these functions utilizes the intrinsic array subscript ranging with the new colon (**:**) operator, which is similar to the one in Matlab®, or simply cite the array name to range over all of its elements. In an OO language like C++, that line would have to be replaced by a formal loop structure block. This intrinsic feature of F90 is used throughout the functionality of this illustrated vector class. Having defined the type Vector, the compiler knows how to evaluate the assignment, "=", of one vector to another. However, it would not have the information for equating a single component vector to a real number. Thus, an overloaded assignment procedure called **equal_Real** has been provided for that common special case. A program to exercise those features of the vector class, along with the validity output as comments, is given in Figure 12 on page 21. A partial extension to a matrix class is shown in Figure 13 on page 22.

```
module class_Vector

! filename: class_Vector.inc

! public, everything by default, but can specify any

  type Vector
     private
     integer                      :: size   ! vector length
     real, pointer, dimension(:) :: data   ! component values
  end type Vector

!              Overload common operators
  interface operator (+)                  ! add others later
     module procedure add_Vector, add_Real_to_Vector ; end interface
  interface operator (-)                  ! add unary versions later
     module procedure subtract_Vector, subtract_Real ; end interface
  interface operator (*)                  ! overload *
     module procedure dot_Vector, real_mult_Vector, Vector_mult_real
  end interface
  interface assignment (=)                ! overload =
     module procedure equal_Real ; end interface
  interface operator (==)                 ! overload ==
     module procedure is_equal_to ; end interface
contains                                  ! functions & operators
function add_Real_to_Vector (v, r) result (new)  ! overload +
  type (Vector), intent(in) :: v
  real,          intent(in) :: r
  type (Vector)             :: new         ! new = v + r
    if ( v%size < 1 ) stop "No sizes in add_Real_to_Vector"
    allocate ( new%data(v%size) ) ; new%size = v%size
 !  new%data = v%data + r ! as array operation, or use implied loop
    new%data(1:v%size) = v%data(1:v%size) + r ; end function

function add_Vector (a, b) result (new)    ! vector + vector
  type (Vector), intent(in) :: a, b
  type (Vector)             :: new         ! new = a + b
    if ( a%size /= b%size ) stop "Sizes differ in add_Vector"
    allocate ( new%data(a%size) ) ; new%size = a%size
    new%data = a%data + b%data    ; end function add_Vector
function assign (values) result (name) ! array to vector constructor
```

```
  real, intent(in) :: values(:)        ! given rank 1 array
  integer          :: length           ! array size
  type (Vector)    :: name             ! Vector to create
    length = size(values); allocate ( name%data(length) )
    name % size = length ; name % data = values; end function assign

function copy_Vector (name) result (new)
  type (Vector), intent(in) :: name
  type (Vector)             :: new
    allocate ( new%data(name%size) ) ; new%size = name%size
    new%data = name%data             ; end function copy_Vector

subroutine  delete_Vector (name)        ! deallocate allocated items
  type (Vector), intent(inout) :: name
  integer                      :: ok    ! check deallocate status
    deallocate (name%data, stat = ok )
    if ( ok /= 0 ) stop "Vector not allocated in delete_Vector"
      name%size = 0 ; end subroutine delete_Vector

function dot_Vector (a, b) result (c)      ! overload *
  type (Vector), intent(in) :: a, b
  real                      :: c
    if ( a%size /= b%size ) stop "Sizes differ in dot_Vector"
      c = dot_product (a%data, b%data) ; end function dot_Vector

subroutine equal_Real (new, R)      ! overload =, real to vector
   type (Vector), intent(inout) :: new
   real,          intent(in)    :: R
     if ( associated (new%data) ) deallocate (new%data)
     allocate ( new%data(1) ); new%size = 1
     new%data = R            ; end subroutine equal_Real

logical function is_equal_to (a, b) result (t_f)  ! overload ==
  type (Vector), intent(in) :: a, b       ! left & right of ==
    t_f = .false.                        ! initialize
    if ( a%size /= b%size ) return       ! same size ?
      t_f = all ( a%data == b%data )     ! and all values match
end function is_equal_to

function length (name) result (n)          ! accessor member
  type (Vector), intent(in) :: name
  integer                   :: n
    n = name % size ; end function length

subroutine list (name)                     ! accessor member
  type (Vector), intent(in) :: name
    print *,"[", name % data(1:name%size), "]"; end subroutine list

function make_Vector (len, values) result(v) ! Optional Constructor
  integer, optional, intent(in) :: len       ! number of values
  real,    optional, intent(in) :: values(:) ! given values
  type (Vector)                 :: v
    if ( present (len) ) then                ! create vector data
      v%size = len ; allocate ( v%data(len) )
      if ( present (values)) then ; v%data = values    ! vector
        else                      ; v%data = 0.d0      ! null vector
      end if ! values present
```



```
    else                                     ! scalar constant
      v%size = 1                  ; allocate ( v%data(1) ) ! default
      if ( present (values)) then ; v%data(1) = values(1)  ! scalar
        else                      ; v%data(1) = 0.d0       ! null
      end if ! value present
    end if ! len present
end function make_Vector

function normalize_Vector (name)  result (new)
  type (Vector), intent(in) :: name
  type (Vector)             :: new
  real                      :: total, nil = epsilon(nil)  ! tolerance
    allocate ( new%data(name%size) ) ; new%size = name%size
    total = sqrt ( sum ( name%data**2 ) )       ! intrinsic functions
    if ( total < nil ) then ; new%data = 0.d0   ! avoid division by 0
      else                  ; new%data = name%data/total
    end if                  ; end function normalize_Vector

subroutine read_Vector (name)              ! read array, assign
  type (Vector), intent(inout) :: name
  integer, parameter           :: max = 999
  integer                      :: length
    read (*,'(i1)', advance = 'no') length
    if ( length <= 0 )   stop "Invalid length in read_Vector"
    if ( length >= max ) stop "Maximum length in read_Vector"
     allocate ( name % data(length) ) ; name % size = length
     read *, name % data(1:length)    ; end subroutine read_Vector

function real_mult_Vector (r, v) result (new) ! overload *
  real,          intent(in) :: r
  type (Vector), intent(in) :: v
  type (Vector)             :: new         ! new = r * v
    if ( v%size < 1 ) stop "Zero size in real_mult_Vector"
    allocate ( new%data(v%size) ) ; new%size = v%size
    new%data = r * v%data         ; end function real_mult_Vector

function size_Vector (name) result (n)     ! accessor member
  type (Vector), intent(in) :: name
  integer                   :: n
    n = name % size ; end function size_Vector

function subtract_Real (v, r) result (new) ! vector-real, overload -
  type (Vector), intent(in) :: v
  real,          intent(in) :: r
  type (Vector)             :: new         ! new = v + r
    if ( v%size < 1 ) stop "Zero length in subtract_Real"
     allocate ( new%data(v%size) ) ; new%size = v%size
     new%data = v%data - r         ; end function subtract_Real

function subtract_Vector (a, b) result (new) ! overload -
  type (Vector), intent(in) :: a, b
  type (Vector)             :: new
    if ( a%size /= b%size ) stop "Sizes differ in subtract_Vector"
     allocate ( new%data(a%size) ) ; new%size = a%size
     new%data = a%data - b%data    ; end function subtract_Vector

function values (name) result (array)        ! accessor member
```



```
    type (Vector), intent(in) :: name
    real                      :: array(name%size)
      array = name % data ; end function values

  function Vector_ (length, values) result(name) ! Public constructor
    integer,      intent(in) :: length           ! array size
    real, target, intent(in) :: values(length)   ! given array
    real, pointer            :: pt_to_val(:)     ! pointer to array
    type (Vector)            :: name             ! Vector to create
    integer                  :: get_m            ! allocate flag
      allocate ( pt_to_val (length), stat = get_m )    ! allocate
      if ( get_m /= 0 ) stop 'allocate error'          ! check
      pt_to_val = values                          ! dereference values
      name      = Vector(length, pt_to_val)       ! intrinsic constructor
  end function Vector_

  function Vector_max_value (a) result (v)       ! accessor member
    type (Vector), intent(in) :: a
    real                      :: v
      v = maxval ( a%data(1:a%size) ); end function Vector_max_value

  function Vector_min_value (a) result (v)       ! accessor member
    type (Vector), intent(in) :: a
    real                      :: v
      v = minval ( a%data(1:a%size) ) ; end function Vector_min_value

  function Vector_mult_real (v, r) result (new) ! vector*real, overload *
    type (Vector), intent(in) :: v
    real,          intent(in) :: r
    type (Vector)             :: new            ! new = v * r
      if ( v%size < 1 ) stop "Zero size in Vector_mult_real"
        new = Real_mult_Vector (r, v) ; end function Vector_mult_real
  end module class_Vector
```

Figure 11: *A Typical Class of Vector Functionality*

```
!        Testing Vector Class Constructors & Operators
include 'class_Vector.f90'                          ! see previous figure
program check_vector_class
  use class_Vector
  type (Vector) :: x, y, z

!          test optional constructors: assign, and copy
  x = make_Vector ()                                  ! single scalar zero
  write (*,'("made scalar x = ")', advance='no'); call list (x)
  call delete_Vector (x) ; y = make_Vector (4)        ! 4 zero values
  write (*,'("made null y = ")',   advance='no'); call list (y)
  z = make_Vector (4, (/11., 12., 13., 14./) )        ! 4 non-zero values
  write (*,'("made full z = ")',   advance='no'); call list (z)
  write (*,'("assign [ 31., 32., 33., 34. ] to x")')
  x = assign( (/31., 32., 33., 34./) )                ! (4) non-zeros
  write (*,'("assigned  x = ")',   advance='no'); call list (x)
  x = Vector_(4, (/31., 32., 33., 34./) )             ! 4 non-zero values
  write (*,'("public    x = ")',   advance='no'); call list (x)
  write (*,'("copy x to y =")', advance='no')
```

```
  y = copy_Vector (x) ; call list (y)                        ! copy

!                test overloaded operators
  write (*,'("z * x gives ")', advance='no'); print *, z*x    ! dot
  write (*,'("z + x gives ")', advance='no'); call list (z+x) ! add
  y = 25.6                                         ! real to vector
  write (*,'("y = 25.6 gives ")',   advance='no'); call list (y)
  y = z                                            ! equality
  write (*,'("y = z gives y as ")',   advance='no'); call list (y)
  write (*,'("logic y == x gives ")', advance='no'); print *, y==x
  write (*,'("logic y == z gives ")', advance='no'); print *, y==z

!                test destructor, accessors
  call delete_Vector (y)                                ! destructor
  write (*,'("deleting y gives y = ")', advance='no'); call list (y)
  print *, "size of x is ", length (x)                     ! accessor
  print *, "data in x are [", values (x), "]"              ! accessor
  write (*,'("2. times x is ")',  advance='no'); call list (2.0*x)
  write (*,'("x times 2. is ")',  advance='no'); call list (x*2.0)
  call delete_Vector (x); call delete_Vector (z)           ! clean up
end program check_vector_class
!  Running gives the output:       ! made scalar x = [0.]
! made null y = [0., 0., 0., 0.]   ! made full z = [11., 12., 13., 14.]
! assign [31., 32., 33., 34.] to x ! assigned x  = [31., 32., 33., 34.]
! public  x = [31., 32., 33., 34.] ! copy x to y = [31., 32., 33., 34.]
! z * x gives 1630.                ! z + x gives   [42., 44., 46., 48.]
! y = 25.6 gives [25.6000004]      ! y = z,   y =  [11., 12., 13., 14.]
! logic y == x gives F             ! logic y == z gives T
! deleting y gives y = []          ! size of x is 4
! data in x : [31., 32., 33., 34.] ! 2. times x is [62., 64., 66., 68.]
! x times 2. is [62., 64., 66., 68.]
```

Figure 12: *Manually Checking the Vector Class Functionality*

```
module class_Matrix                       ! file: class_Matrix.f90
type Matrix
     private
     integer       :: rows, columns ! matrix sizes
     real, pointer :: values(:,:)   ! component values
end type Matrix !              Overload common operators

  interface operator (+)
     module procedure Add_Matrix, Add_Real_to_Matrix ; end interface
  . . .
contains    ! constructors, destructors, functions & operators

!                -- constructors & destructors --
function Matrix_ (rows, columns, values) result(M) ! Public constructor
  integer,      intent(in) :: rows, columns         ! array size
  real, target, intent(in) :: values(rows, columns)  ! given array
  real, pointer            :: pt_to_val(:, :)       ! pointer to array
  type (Matrix)            :: M                     ! Matrix to create
    pt_to_val => values                          ! point at array
```

```
    M         = Matrix(rows, columns, pt_to_val) ! intrinsic
constructor
    active    = active + 1                       ! increment activity
end function Matrix_

. . .

function Add_Matrix (a, b) result (new) ! matrix + matrix, overload +
  type (Matrix), intent(in) :: a, b     ! left and right of +
  type (Matrix)             :: new      ! new = a + b
    if ( a%rows /= b%rows .or. a%columns /= b%columns ) stop &
        "Error: Sizes differ in Add_Matrix"
    allocate ( new%values(a%rows, a%columns) )
    new%rows   = a%rows   ; new%columns = a%columns      ! sizes
    new%values = a%values + b%values   ! intrinsic array addition
end function Add_Matrix
```

Figure 13: *Segments of a Typical Matrix Class*

4. Conclusion

There are dozens of OOP languages. Persons involved in engineering computations need to be aware that F90 can meet almost all of their needs for a OOP language. At the same time it includes the F77 standard as a subset and thus allows efficient use of the many millions of Fortran functions and subroutines developed in the past. The newer F95 standard is designed to make efficient use of super computers and massively parallel machines. It includes most of the High Performance Fortran features that are in wide use. Thus, efficient use of OOP on parallel machines is available through F95. None of the OOP languages have all the features one might desire. For example, the useful concept of a "template" which is standard in C++ is not in the F90 standard. Yet the author has found that a few dozen lines of F90 code will define a preprocessor that allows templates to be defined in F90 and expanded in line at compile time. The real challenge in OOP is the actual OO analysis and OO design (Coad, 1991; Rumbaugh, 1991) that must be completed before programming can begin, regardless of the language employed. For example, several authors have described widely different approaches for defining classes to be used in constructing OO finite element systems (e.g., Barton, 1994; Filho, 1991; Machiels, 1997). These areas still merit study and will be important to the future of engineering computations. Those programmers still employing F77 should try the OO benefits of F90 and F95 as one approach for improving the efficiency of their computations.

5. References

1. J. C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith and J.L. Wagener, Fortran 90 Handbook, McGraw Hill, 1992.

2. J. E. Akin, "A RP-Adaptive Scheme for the Finite Element Analysis of Linear Elliptic Problems", Mathematics of Finite Elements and Applications: 1996, J. R. Whiteman (Ed.), Academic Press, pp. 427-438, 1997.

3. J.J. Barton and L.R. Nackman, Scientific and Engineering C++, Addison Wesley, 1994.

4. P. Coad and E. Yourdon, Object Oriented Design, Prentice Hall, 1991.

5. Y. Dubois-P`elerin and T. Zimmermann, "Object-oriented finite element programming: III. An efficient implementation in C++" Comp. Meth. Appl. Mech. Engr., Vol. 108, pp. 165-183, 1993.

6. Y. Dubois-P`elerin and P. Pegon, "Improving Modularity in Object-Oriented Finite Element Programming," Communications in Numerical Methods in Engineering, Vol. 13, pp. 193-198, 1997.

7. J. S. R. A. Filho and P. R. B. Devloo, "Object Oriented Programming in Scientific Computations," Engineering Computations, Vol. 8, No. 1, pp. 81-87, 1991.

8. J. R. Hubbard, Programming with C++, McGraw Hill, 1994.

9. L. Machiels and M. O. Deville, "Fortran 90: On Entry to Object Oriented Programming for the Solution of Partial Differential Equations," ACM Trans. Math. Software, Vol. 23, No. 1, pp. 32-49, Mar. 1997.

10. W. H. Press, S. A. Teukolsky, W. T. Vettering and B. P. Flannery, Numerical Recipes in Fortran 90, Cambridge Press, 1996.

11. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Object Oriented Modeling and Design, Prentice Hall, 1991.

Matlab is a registered trademark of The MathWorks, Inc.

