Introduction to Object-Oriented Concepts using Fortran90

Viktor K. Decyk

Department of Physics and Astronomy
University of California, Los Angeles
Los Angeles, CA 90095-1547
&
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109-8099

email: decyk@physics.ucla.edu

Charles D. Norton* and Boleslaw K. Szymanski

Department of Computer Science
and
Scientific Computation Research Center (SCOREC)
Rensselaer Polytechnic Institute
Troy, NY 12180-3590

email: szymansk@cs.rpi.edu

*Current address: Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109-8099
email: nortonc@olympic.jpl.nasa.gov

Abstract

Fortran90 is a modern, powerful language with features that support important new programming concepts, including those used in object-oriented programming. This paper explains the concepts of data encapsulation, function overloading, classes, objects, inheritance, and dynamic dispatching, and how to implement them in Fortran90. As a result, a methodology can be developed to do object-oriented programming in the language.

I. Introduction

Fortran, still the most widely used scientific programming language, has evolved every 10 years or so to incorporate the most recent, proven, ideas which have emerged from computer science and software engineering. The latest version, Fortran90, has incorporated a great many new ideas, but scientific programmers generally are aware of only one of them: array syntax. These other new ideas permit better and safer programming design. Furthermore, they allow the software application to be expressed in terms of familiar and appropriate scientific concepts. These new capabilities in Fortran90 make scientific programs easier to understand, modify, share, explain and extend. As a result, much more ambitious programming problems can be attacked in a manageable way.

There are several properties of program design that are useful in the programming and maintenance of large computational projects. First, the code which modifies a given data structure should be localized or confined to one location in the program and not spread, uncontrollably, throughout the program. This property is called an *encapsulation*. In a sense, it is a generalization of the familiar notion of a function or subroutine. However, modern encapsulation allows all related operations to be grouped together around a data type. For example, given a data type representing a logical, encapsulation may group together a set of operations that perform logical operations. A related notion is that of *information hiding*. Once all the operations are encapsulated, one can hide the details by which the results are computed, allowing only well defined operations to be applied to the data. For example, one can use the .not. operation on the intrinsic type logical, without worrying about the way the compiler writers implemented the operation or the internal representation of logicals. The divide operation on logicals, on the other hand, is not defined in Fortran and therefore cannot be used.

Using encapsulation and information hiding, the users can define their own data types, called *abstract data types*. For example, one can define a data type representing an angle. Abstract data types are further defined by the operations that can be applied to them. A *class definition* encapsulates the code for the allowable operations, or *methods*, written by the programmer along with the abstract data type, hiding the implementation details there. For instance, users of the data type *angle* can apply the sine and cosine functions to *angle* variables (*objects*) if such methods have been defined for *angles*, but cannot multiply two *angles* if such a method has not been defined. Encapsulated code can be safely changed (for example, a different method of computing sines could be used on a new architecture) without the need to change the code using these functions. The other advantage of abstract data types is that the software designer can describe the program in terms resembling the application domain. For example, if a data type *electron* is defined, one could define what it means to push an *electron* object a given distance in space. (The *electron* is pushed in this view, whereas in Fortran77 the values of arrays representing the electron coordinates are updated).

Some operations are defined for more than one standard type. For example multiplication can be defined for integer arguments, real arguments, vectors and arrays. This property is called *overloading* and the concept extends to abstract data types. Consider an *electron* type and an *ion* type, both of which permit a push operation to be applied, but using a different procedure in each case. Overloading helps to write shorter and clearer programs by providing a general operation in which the specific action taken is defined by the type of the arguments at compile-time.

Often in science, properties of some entities are abstracted and grouped into classes. For example, *electron* and *ion* types can be considered different kinds of *species* types. If separate data types were created for *electron* and *ion*, many operations would be shared by these two types. Overloading can help, but it is even more convenient to introduce the abstract data type *species* for the common properties. Then, like in everyday life, everything that applies to the *species* type can also be applicable to *electron* and *ion* types (but not the opposite). Once the data type *species* is defined one can simplify the description of *ion* and *electron* types by permitting *inheritance* of *species* properties by those two more specialized data types.

All of the ideas presented so far are part of the modern programming paradigm called object-oriented programming (OOP). Many users with large investments in Fortran77 have reason to be reluctant to shift to a very different programming style and language. The demands on development time and the training required to shift to a new approach, combined with the many years invested in existing Fortran77 based applications and the need to continually produce new science are good reasons to be skeptical of experimenting with new ideas, as promising as they might appear.

Fortran90 supports these new important programming concepts, including those used in object-oriented programming. Since Fortran90 is backward compatible with Fortran77, it is possible to incorporate these new ideas into old programs in an incremental fashion, enabling the scientist to continue his or her scientific activities. Some of these ideas are useful for the typical kinds of programs written by individual authors now. The usefulness of other ideas only becomes apparent for more ambitious programs written by multiple authors. These are programs that might never have been written in Fortran77 because the complexity involved would have been unmanageable. These new ideas enable more productive program development, encourage software collaboration and allow the scientist to use the same abstract concepts in a program that have been used so successfully in scientific theory. Scientific productivity will then improve. Additionally, there is also a migration path to parallel computers, since High Performance Fortran (HPF) is also based on Fortran90.

In this paper, we will introduce the concepts of data encapsulation, function overloading, classes and objects, inheritance, and dynamic dispatching in Fortran90. Since these are the fundamental building blocks of object-oriented programming, it is important to understand them before one can effectively use OOP in Fortran90. Many of these ideas are powerful by themselves and are useful even without adopting the object-oriented paradigm. This paper is intended to be introductory in nature. For those who wish to pursue these ideas further, there are a number of references which discuss object-oriented ideas in a language independent manner [1-3]. There are many textbooks available on Fortran90 and C++. Two that we have found useful are those by Ellis [4] and Lippman [5].

II. Data Encapsulation with Array Objects

The first concept we will discuss is data encapsulation, which means that only those procedures which need to have access to certain data are aware of it. To illustrate how data encapsulation works in Fortran90, consider an example from a real to complex Fast Fourier Transform (FFT) subroutine written in Fortran77. The *interface* to the procedure, that is, the list of arguments and their types, is defined as follows:

```
      subroutine fft1r(f,t,isign,mixup,sct,indx,nx,nxh)
      integer isign, indx, nx, nxh, mixup(nxh)
      real f(nx)
      complex sct(nxh), t(nxh)
c rest of procedure goes here
      return
      end
```

Here `f` is the array to be transformed (and the output), `t` is a temporary work array, `mixup` is a bit-reverse table, `sct` is a sine/cosine table, `indx` is the power of 2 defining the length of the transform, and `nx` (`>=2**indx`) is the size of the `f` array, and `nxh` (`=nx/2`) is the size of the remaining arrays. The variable `isign` determines the direction of the transform (or if zero, initiates the tables `mixup` and `sct`.)

Since the procedure `fft1r` is designed to work with variable size data (and might be compiled separately), and Fortran77 cannot dynamically allocate such data, the work and table arrays `t`, `sct`, and `mixup` have to be declared in the main program and passed to the procedure. If the FFT procedure is itself embedded inside other procedures, then all these arrays have to be passed down the chain of arguments or else stored in a common block. Thus the main program might look something like:

```
      program main
      integer isign, indx, nx, nxh
      parameter(indx=11,nx=2**indx,nxh=nx/2)
      integer mixup(nxh)
      real f(nx)
      complex sct(nxh), t(nxh)
c initialize fft tables
      isign = 0
      call fft1r(f,t,isign,mixup,sct,indx,nx,nxh)
      stop
      end
```

The goal of data encapsulation is make the FFT call look like:

```
      call fft1r(f,isign)
```

where all the auxiliary arrays and constants which are needed only by the FFT are hidden inside the FFT, and the rest of the program does not have to be concerned about them. This hiding of data greatly simplifies bookkeeping with procedures.

Fortran90 allows dynamic arrays which are only used inside a procedure to be created and destroyed there, and they are therefore unknown outside the procedure. One mechanism for such encapsulation is the automatic array, which is created on entry and destroyed upon exiting a procedure. It is easy to implement the work array `t` as an automatic array. One just omits the name from the argument list:

```
      subroutine fft1r(f,isign,mixup,sct,indx,nx,nxh)
      integer isign, indx, nx, nxh, mixup(nxh)
      real f(nx)
      complex sct(nxh)
! t is an automatic array, it disappears on exit
      complex, dimension(nxh) :: t
! rest of procedure goes here
      end subroutine fft1r
```

Notice that we have begun to use the new Fortran 90 :: syntax for declaring arrays, and a new END SUBROUTINE statement. Automatic arrays differ from local arrays previously available in Fortran77 because their dimensions can now be variables.

Another mechanism for encapsulation of arrays in Fortran90 is the allocatable (or deferred-size) array. They are similar to automatic arrays, except that their creation (allocation) and destruction (deallocation) are entirely under programmer control. Such arrays are created with the ALLOCATE statement. If the SAVE attribute is used in their declaration, then they will not be destroyed on exit from the procedure. They can be explicitly destroyed with the DEALLOCATE statement.

For now, let us assume that the table arrays `mixup` and `sct` do not change between calls to the FFT. We can then remove them from the argument list in our example and explicitly ALLOCATE them inside the procedure when the tables are initialized, as follows:

```
      subroutine fft1r(f,isign,indx,nx,nxh)
      integer isign, indx, nx, nxh
      real f(nx)
! mixup and sct are saved, allocatable arrays
      integer, dimension(:), allocatable, save :: mixup
      complex, dimension(:), allocatable, save :: sct
! t is an automatic array
      complex, dimension(nxh) :: t
! special initialization call
      if (isign.eq.0) allocate(mixup(nxh),sct(nxh))
! rest of procedure goes here
      end subroutine fft1r
```

Later, we will add error checking conditions.

A very powerful feature of Fortran90 is that arrays are actually array objects which contain not only the data itself, but information about their size. This was previously only available for character arrays in Fortran77, which supplied the LEN intrinsic to obtain the character length. In Fortran90, assumed-shape arrays are available whose dimensions can be obtained from the SIZE intrinsic. This is a third mechanism useful for data encapsulation. They are declared like ordinary arrays, except their dimension lengths are replaced with a colon. We can now omit the dimensions `nx` and `nxh` from the argument list, and obtain them inside the procedure. The declaration of the automatic array `t` also has to be revised to use the SIZE intrinsic. The result is:

```
      subroutine fft1r(f,isign,indx)
! f is an assumed-shape array
      real, dimension(:) :: f
```

```
      integer :: isign, indx, nx, nxh
      integer, dimension(:), allocatable, save :: mixup
      complex, dimension(:), allocatable, save :: sct
! t is an automatic array whose size is determined from array f
      complex, dimension(size(f)/2) :: t
! size of arrays mixup and sct are determined from array f.
      nx = size(f)
      nxh = nx/2
! special initialization call
      if (isign.eq.0) allocate(mixup(nxh),sct(nxh))
! rest of procedure goes here
      end subroutine fft1r
```

In order to use assumed-shape arrays the compiler must have knowledge of the actual argument types being used when the procedure is called. One way this information can be supplied is with the INTERFACE statement, which declares to the main program the argument types of the procedure. For example, one can write:

```
      program main
      integer :: isign
      integer, parameter :: indx=11, nx=2**indx
      real, dimension(nx) :: f
! declare interface
      interface
         subroutine fft1r(a,i,j)
         real, dimension(:) :: a
         integer :: i, j
         end subroutine fft1r
      end interface
! initialize fft tables
      isign = 0
      call fft1r(f,isign,indx)
      stop
      end
```

where we have used the new form of the PARAMETER statement. An additional advantage of explicit INTERFACE blocks is the compiler will now check and require that the actual arguments passed to the procedure match in number and type with those declared in the interface. Thus if one accidentally omits the argument `isign` in a procedure call

```
      call fft1r(f,indx)
```

the compiler will flag this.

One possible source of error is that one can mistakenly declare the data in an INTERFACE block to be different than the actual data in the procedure. This source of error is removed if the procedure is stored in a MODULE which is then "used," because in this case the compiler creates the INTERFACE automatically. The USE statement is similar to the INCLUDE extension commonly found in Fortran77, but it is not a text substitution. Rather, it makes information "available", and is much more powerful than the INCLUDE statement. Thus:

```
      module fft1r_module
```

```
      contains
         subroutine fft1r(f,isign,indx)
         real, dimension(:) :: f
         integer :: isign, indx, nx, nxh
         integer, dimension(:), allocatable, save :: mixup
         complex, dimension(:), allocatable, save :: sct
         complex, dimension(size(f)/2) :: t
         nx = size(f)
         nxh = nx/2
! special initialization call
         if (isign.eq.0) allocate(mixup(nxh),sct(nxh))
! rest of procedure goes here
         end subroutine fft1r
      end fft1r_module
!
      program main
      use fft1r_module     ! explicit interface not needed now
      integer :: isign = 0
      integer, parameter :: indx=11, nx=2**indx
      real, dimension(nx) :: f
! initialize fft tables
      call fft1r(f,isign,indx)
      stop
      end
```

where we have used a new way to initialize the integer `isign`.

Fortran90 supports a number of other statements and attributes that contribute to programming safety. One is the IMPLICIT NONE statement that requires all variables to be explicitly declared. Another is the INTENT attribute for arguments, that declares whether arguments are intended as input only, output only, or both. If we declare the arguments in `fft1r` as follows:

```
         subroutine fft1r(f,isign,indx)
         implicit none
         real, dimension(:), intent(inout) :: f
         integer, intent(in) :: isign, indx
```

then the variables `isign` and `indx` cannot be modified in this procedure since they were declared with INTENT(IN) attributes. Such features mean that more errors are now caught by the compiler rather than by the operating system when the code is running. Because of this added safety we will use modules for all of the remaining subroutines in this paper.

The encapsulation of data we have illustrated in this example makes it easier for multiple authors to independently develop programs which will be used by others. The FFT program now has a simple interface which is less likely to be changed, so that even if the author of the procedure makes changes internally, users of the procedure do not have to change their code.

Another benefit of such an approach is that one can hide old, ugly code that cannot be changed, perhaps because one does not have access to the source. For example, if one were using a library FFT which was optimized for some specific architecture, one could encapsulate it in a “shell” procedure which allocates any work or table arrays needed and then calls the library FFT. Since the details of the library FFT are hidden from the user, one could replace it with another by making changes only in this "shell" procedure and not impact the rest of the code. This allows a code to remain portable and yet optimized.

Let us now add more error checking when allocating data. The ALLOCATE statement allows an optional error return code to check if the data was actually allocated. And the ALLOCATED statement allows one to check if the data is already allocated (perhaps we had previously used the FFT with different length data). Thus the following version of the allocation is safer:

```
if (isign.eq.0) then
   if (allocated(mixup)) deallocate(mixup)
   if (allocated(sct)) deallocate(sct)
   allocate(mixup(nxh),sct(nxh),stat=ierr)
   if (ierr.ne.0) then
      print *,'allocation error'
      stop
   endif
endif
```

One can simplify the interface even further by noting that the FFT length parameter `indx` is only needed during the initialization of the FFT. In subsequent calls it would be an error to use a different value without initializing new tables. Fortran90 supports the use of OPTIONAL arguments and the intrinsic PRESENT to determine if it was passed. With these tools, we can save the `indx` parameter passed during initialization and not require it to be passed subsequently. Furthermore, if we initialize the saved `indx` parameter to some nonsense value, we can test it subsequently to prevent the FFT from being used before the FFT tables were initialized. The result is:

```
subroutine fft1r(f,isign,indx)
integer, intent(in), optional :: indx
integer, save :: saved_indx = -1  ! initialize to nonsense
if (isign.eq.0) then
   if (.not.present(indx)) then
      print *,'indx must be present during initialization!'
      stop
   endif
   saved_indx = indx
else
   if (saved_indx.lt.0) then
      print *,'fft tables not initialized!'
      stop
   endif
endif
```

In the following version of the FFT, the procedure `lib_fft1r` is intended to refer to some library FFT where either the source code is unavailable or one does not desire to change it. This new version is much safer and easier to use and modify.

```
      subroutine fft1r(f,isign,indx)
      implicit none
      real, dimension(:), intent(inout) :: f
      integer, intent(in) :: isign
      integer, intent(in), optional :: indx
      integer :: nx, nxh, ierr
      integer, save :: saved_indx = -1
      integer, dimension(:), allocatable, save :: mixup
      complex, dimension(:), allocatable, save :: sct
      complex, dimension(size(f)/2) :: t
      nx = size(f)
      nxh = nx/2
! special initialization call
      if (isign.eq.0) then
! indx must be present during initialization
         if (.not.present(indx)) then
            print *,'indx must be present during initialization'
            stop
         endif
! indx must be non-negative
         if (indx.lt.0) then
            print *,'indx must be non-negative'
            stop
         endif
! save indx for future calls
         saved_indx = indx
! deallocate if already allocated
         if (allocated(mixup)) deallocate(mixup)
         if (allocated(sct)) deallocate(sct)
! allocate table arrays
         allocate(mixup(nxh),sct(nxh),stat=ierr)
! check if allocation error
         if (ierr.ne.0) then
            print *,'allocation error'
            stop
         endif
! make sure fft tables initialized
      else
         if (saved_indx.lt.0) then
            print *,'fft tables not initialized!'
            stop
         endif
      endif
! call old, ugly but fast fft here
! saved_indx used here instead of indx
      call lib_fft1r(f,t,isign,mixup,sct,saved_indx,nx,nxh)
      end subroutine fft1r
```

During initialization one would call:

```
      call fft1r(f,isign,indx)
```

But subsequently one can use just:

```
      call fft1r(f,isign)
```

Logically, we have bundled two distinct operations, initialization and performing the FFT, into one procedure. There is a third procedure which would be useful, deallocating the internal table arrays to free up memory if we are done with performing FFTs. This could be done by adding an extra, OPTIONAL argument to the procedure, and adding more code, but this is unattractive. It makes for clearer programming to separate logically distinct operations into distinct procedures. The difficulty with this is how to allow distinct procedures to share access to the internal table arrays `mixup` and `sct`. In Fortran77, the only mechanism to do this was common blocks. In Fortran90, there is a new mechanism: modules can contain global data which are shared by all the procedures in the module without explicitly declaring them inside the procedures. This is a new idea in Fortran, although common in other languages such as C++. Furthermore, this global data can be made local to the module and inaccessible to other procedures which USE the module.

In our example, we will make the table arrays `mixup` and `sct`, as well as the integer `saved_indx` global by moving their declaration outside the procedure to the declaration section at the beginning of the module. We will also add the PRIVATE attribute, to block access to this data from outside the module. Adding the deallocation procedure, the module looks like:

```
      module fft1r_module
! all module procedures have access to this data
      integer, save, private :: saved_indx = -1
      integer, dimension(:), allocatable, save, private :: mixup
      complex, dimension(:), allocatable, save, private :: sct
      contains
         subroutine fft1r_end
! this procedure has access to saved_indx, mixup, and sct
! reset saved_indx to nonsense value
         saved_indx = -1
! deallocate table arrays
         deallocate(mixup,sct)
         end subroutine fft1r_end
! other procedures go here
      end module fft1r_module
```

By separating the original `fft1r` procedure into a new initialization (`fft1r_init`) and FFT procedure, we no longer need to use optional arguments. In the final version of this module which is shown in Appendix A, we have used the '`;`' syntax which allows multiple statements on one line.

Allocatable arrays can also be used in the main program, which allows one to create all arrays at run time rather than at compile time. In Fortran90, one no longer has to recompile a code because the dimensions change. (It is the programmer's responsibility, however, not to use an allocatable array before it has been allocated or after it has been deallocated.) In the following main program, we make `f` an allocatable array, obtain the value of `indx` from the input device, allocate `f`, and use the new array constructor syntax to initialize it.

```
      program main
      use fft1r_module
      implicit none
      integer :: indx, nx, i
      real, dimension(:), allocatable :: f
! write prompt without linefeed
      write (6,'(a)',advance='no') 'enter indx: '
! obtain indx from input device
      read (5,*) indx
! allocate array f
      nx = 2**indx
      allocate(f(nx))
! initialize data using array constructor
      f = (/(i,i=1,nx)/)
! initialize fft
      call fft1r_init(indx)
! call fft
      call fft1r(f,-1)
! terminate fft
      call fft1r_end
      stop
      end
```

Notice that we have not modified the original `lib_fft1r` procedure, which is a private procedure in the module. Instead we have simplified the user interface to the bare essentials while adding substantial safety to its usage.

III. Function Overloading

Function overloading refers to using the same function name but performing different operations based on argument type. Fortran77 intrinsic functions and operators have always had this feature. For example, the divide '/' symbol gives different results depending on whether the operands are integers, reals, or complex variables. Similarly, the intrinsic procedure REAL(a) will convert an integer to a real, if `a` is an integer, but will return the real part of `a`, if `a` is complex.

In Fortran90, generic functions allow user defined functions to also have this feature. In the case of the FFT example, users of Fortran77 have had to remember to use different function names for every possible type of FFT, such as real to complex, complex to complex, 1 dimensional, 2 dimensional, single precision or double precision FFTs. The generic function facility allows a single name, for example, `fft`, to be used for all of them, and the compiler will automatically select the correct FFT to use based on the number and types of arguments actually used.

Thus in the case of the 1d real to complex FFT, we had a procedure with the following interface:

```
subroutine fft1r(f,isign)
real, dimension(:), intent(inout) :: f
integer, intent(in) :: isign
```

In a manner similar to what we described in the previous section, one can construct a 2d real to complex FFT with the following interface:

```
subroutine fft2r(f,isign)
real, dimension(:,:), intent(inout) :: f
integer, intent(in) :: isign
```

where the procedure `fft2r` "hides" an old, ugly but fast 2d real to complex FFT which has lots of arguments. In the first case the argument `f` is a real, one dimensional array, while in the second case it is a real, two dimensional array. If both of these procedures are in the same module, one constructs a generic function `fft` by placing the following statements in the declaration section of the module:

```
interface fft
   module procedure fft1r
   module procedure fft2r
end interface
```

Then in a main program which uses the module, the statement

```
call fft(f,isign)
```

will call the procedure `fft1r`, if `f` is a real, one dimensional array, or will call `fft2r`, if `f` is a real, two dimensional array. If `f` is any other type of argument, a compile error will be generated.

If the 2 dimensional FFT is contained in a separate module, then two separate INTERFACE statements are required. In the first module one includes

```
interface fft
```

```
      module procedure fft1r
   end interface
```

and in the second module one includes

```
   interface fft
      module procedure fft2r
   end interface
```

The main program then uses both modules, and each module procedure will then be added to the list of procedures that have the generic interface `fft`.

In a similar manner, one can include in the same interface all other types of FFTs and the programmer is protected from making errors in calling the wrong procedure. If desired, one can even make the specific names `fft1r`, `fft2r` inaccessible by adding the declaration:

```
   private :: fft1r, fft2r
```

in the module. One advantage of this is that the specific names can be reused in other modules without conflict. Function overloading is also called ad hoc polymorphism.

IV. Derived Types, Classes, and Objects

Fortran has a number of intrinsic data types, such as integer, real, complex, logical, and character, for which operators and functions are defined in the language. An important new feature of Fortran90 is user defined data types. A user defined type, also known as an abstract data type, is called a derived type in Fortran90. It is built up from intrinsic types and previously defined user types. One simple use of this new capability is to bundle together various scalars that normally get passed together as arguments to procedures. For example, consider the following interface from a particle pushing subroutine written in Fortran77:

```
      subroutine push1 (part,fx,qbm,dt,ek,np,idimp,nop,nx)
      integer np, idimp, nop, nx
      real qbm, dt, ek, part(idimp,nop), fx(nx)
c rest of procedure goes here
      return
      end
```

Here `part` is the array which contains the particle coordinates and velocities and `fx` is the electric field array. The integer `idimp` is the dimensionality of phase space, `nop` is the maximum number of particles allowed, and `nx` is the size of the electric field array. As we saw in the FFT example, we do not have to pass these integers in Fortran90, since they can be determined by the SIZE intrinsic if `part` and `fx` are passed as assumed-shape arrays. The integer `np` (`np<=nop`) is the actual number of valid particles in the `part` array, `qbm` is the charge/mass ratio, `ek` is the kinetic energy of the `np` valid particles, and `dt` is the time step. All of the scalars except for the time step describe a group of charged particles and they usually are passed together whenever the particles are processed by some procedure. We can use a derived type to store them together as follows:

```
      type species_descriptor
         integer :: number_of_particles
         real :: charge, charge_to_mass, kinetic_energy
      end type species_descriptor
```

This is similar to structures and record types which appear in other programming languages. We have added charge to the list, since there are some procedures which also require that. Notice that in this derived type, there are components of both integer and real type, so that this could not have been implemented with just an array in Fortran77. To create a variable of this type, one makes the following declaration:

```
      type (species_descriptor) :: electron_args, ion_args
```

where we have created two variables of type `species_descriptor`, one for electrons and one for ions. The components of this new type are accessed with the '%' symbol. Thus we can assign values as follows:

```
      electron_args%number_of_particles = 1000
      electron_args%charge = 1.0
```

It is best to put the definition in the declaration section of a module along with the new `push1` subroutine (to avoid having to declare an explicit interface for it), as shown in

Appendix B. Then this module is "used" in the main program to give access to the derived type and new push1 procedure, which can now be called with a much simpler interface:

```
      call push1(part,fx,electron_args,dt)
```

We have shown here a simple use of derived types, merely to reduce bookkeeping when passing arguments to procedures. But derived types are much more powerful than that. They can be used to express sophisticated, abstract quantities. In fact, with derived types it is possible to express in programming the same high level, abstract quantities that physicists are used to in their mathematics. To illustrate how one might begin to express more sophisticated mathematics in programming, let us define a new private_complex type and the procedures which will operate on that type. This is, of course, an academic exercise for Fortran programmers, since the complex type already exists in the language. Nevertheless, it is a useful example to illustrate the basic principles involved and will lead to our definition of classes. This type is defined as follows:

```
      type private_complex
         real :: real, imaginary
      end type private_complex
```

To create variables a, b, and c of this new type, and assign values, one proceeds as before:

```
      type (private_complex) :: a, b, c
! assign values to a
      a%real = 1.0
      a%imaginary = 2.0
```

If this private_complex type behaves the same as ordinary complex numbers, then multiplication of c = a*b can be defined as follows:

```
      c%real = a%real*b%real - a%imaginary*b%imaginary
      c%imaginary = a%real*b%imaginary + a%imaginary*b%real
```

A new function pc_mult to multiply private_complex numbers could then be written:

```
      type (private_complex) function pc_mult(a,b)
      type (private_complex), intent(in) :: a, b
      pc_mult%real = a%real*b%real - a%imaginary*b%imaginary
      pc_mult%imaginary = a%real*b%imaginary + a%imaginary*b%real
      end function pc_mult
```

Note that this function returns a variable of type `private_complex`. One can thus multiply two numbers of this type with the following statement:

```
      c = pc_mult(a,b)
```

It makes sense to place a new derived type together with the procedures which operate on that type into the same module:

```
      module private_complex_module
! define private_complex type
      type private_complex
         real :: real, imaginary
      end type private_complex
      contains
       type (private_complex) function pc_mult(a,b)
! multiply private_complex variables
       type (private_complex), intent(in) :: a, b
       pc_mult%real = a%real*b%real - a%imaginary*b%imaginary
       pc_mult%imaginary = a%real*b%imaginary + a%imaginary*b%real
       end function pc_mult
      end module private_complex_module
```

A program to illustrate the multiplication of two `private_complex` numbers then looks like the following:

```
      program main
! bring in private_complex definition and procedures
      use private_complex_module
! define sample variables
      type (private_complex):: a, b, c
! initialize sample variables
      a%real = 1. ; a%imaginary = -1.
      b%real = -1. ; b%imaginary = 2.
! perform multiplication
      c = pc_mult(a,b)
      print *,'c=', c%real, c%imaginary
      stop
      end program main
```

It is also possible to encapsulate the individual components of a derived type. This is a common and useful practice in object-oriented programming. It means that when a module is "used" in another program unit, the `private_complex` type can be defined, but the individual components, such as a%real or a%imaginary are not accessible. In the sample program above, the individual components were accessed in initializing the data and in printing the result of the multiplication. If the components are encapsulated, then additional procedures would have to be provided in the module to perform this function. The encapsulation is achieved by adding the PRIVATE attribute to the derived type definition as follows:

```
      type private_complex
         private
         real :: real, imaginary
      end type private_complex
```

A procedure to initialize a `private_complex` number from real numbers can be written as follows:

```
      subroutine pc_init(a,real,imaginary)
! initialize private_complex variable from reals
      type (private_complex), intent(out) :: a
      real, intent(in) :: real, imaginary
      a%real = real
      a%imaginary = imaginary
      end subroutine pc_init
```

while one to display the contents can be written:

```
      subroutine pc_display(a,c)
! display value of private_complex variable with label
      type (private_complex), intent(in) :: a
      character*(*), intent(in) :: c
      print *, c, a%real, a%imaginary
      end subroutine pc_display
```

The main program then looks like the following:

```
      program main
      use private_complex_module
      type (private_complex) :: a, b, c
! initialize sample variables
      call pc_init(a,1.,-1.)
      call pc_init(b,-1.,2.)
! perform multiplication
      c = pc_mult(a,b)
! display result
      call pc_display(c,'c=')
      stop
      end program main
```

The advantage of such encapsulation is that procedures in other modules can never impact the internal representation of the `private_complex` type. Furthermore, any changes made to the internal representation of `private_complex` type would be confined to this module, and would not impact program units in other modules. This makes it easier to develop software with interchangeable parts.

We have seen earlier how functions can be overloaded. In Fortran90, operators such as `'*'` can also be overloaded. This is also done with the INTERFACE statement, which is placed in the declaration section of the module, as follows:

```
      interface operator(*)
         module procedure pc_mult
      end interface
```

We have now equated the operator '*' with the name `pc_mult`. Thus in the main program, one can multiply two `private_complex` numbers using the more familiar syntax:

```
      c = a*b
```

If one adds the declaration:

```
      private :: pc_mult
```

to the module, one can also make the original name `pc_mult` no longer accessible.

In the language of object-oriented programming, the module we have just created is known as a class. It consists of a derived type definition, known as a class name, along with the procedures which operate on that class, called class member functions. The components of the derived type are called the class data members, while global data in the module (if any) corresponds to static class data members. The actual variable of type `private_complex` is known as an object.

To make this appear more familiar to those who already know C++, we will adopt the convention to make the derived type the first argument in all the module procedures, and we will give it the name "`this`." We will also overload the initialization function `pc_init` and give it the public name "`new`," while making the specific name `pc_init` private. The final version of the `private_complex` class is listed in Appendix C. In this version we have provided a new public name "`display`" to the procedure `pc_display`, but we have not made it private, so both names can still be used.

The main program which uses this class can be written:

```
      program main
      use private_complex_class
      type (private_complex) :: a, b, c
      call new(a,1.,-1.)
      call new(b,-1.,2.)
! perform multiplication
      c = a*b
      call pc_display(c,'c=')
      stop
      end program main
```

V. Inheritance

We have seen how to create simple classes in Fortran90 by storing together derived type definitions and procedures operating on that type which can then be "used" in other program units, including other modules. Inheritance, in the most general sense, can be defined as the ability to construct more complex (derived) classes from simpler (base) classes in a hierarchical fashion. Inheritance has been also used as a term that describes how this idea is implemented in a particular language, which results in as many definitions as there are languages. We will discuss two methods of implementing this idea which are useful in Fortran90, a general one and a more specialized but common one.

We will begin with the most general case, where one class makes use of another. Such a form of inheritance is sometimes called class composition, since classes are composed of other classes. To illustrate this, let us create a new class which consists of `private_complex` arrays. Array syntax for intrinsic data types is a powerful new feature of Fortran90, and it is instructive to see how one can implement it for user defined types. Fortran90 permits one to define arrays of derived types as follows:

```
program main
use private_complex_class
integer, parameter :: nx = 8
type (private_complex), dimension(nx) :: f
```

We can initialize the elements of this array type by calling in a loop the "`new`" procedure we defined in the `private_complex_class`:

```
do i = 1, nx
   call new(f(i),real(i),-real(i))
enddo
```

Note that the do loop without statement numbers is now an official part of Fortran90. In this way, we can create a procedure called `pcarray_init` which will convert two real arrays into an array of type (private_complex). We place this procedure in a new module called `complex_array_class`, as follows:

```
      module complex_array_class
! get private_complex type and its public procedures
      use private_complex_class
      contains
         subroutine pcarray_init(this,real,imaginary)
! initialize private_complex array from real arrays
         type (private_complex), dimension(:), intent(out) :: this
         real, dimension(:), intent(in) :: real, imaginary
         do i = 1, size(this)
! new here uses the pc_init procedure
            call new(this(i),real(i),imaginary(i))
         enddo
         end subroutine pcarray_init
      end module complex_array_class
```

In this example, the `private_complex` module is "used" (or inherited) by the `complex_array` module. This means that all the public entities in the first module (the base class) will be available to the second module (the derived class). Since arrays of derived type are considered a different type than a single scalar of that type, we can include the following interface statement in the module to overload the procedure `new` so that it can also execute `pcarray_init`:

```
      interface new
         module procedure pcarray_init
      end interface
```

Now, if the argument of `new` is a scalar of type `private_complex`, then `pc_init` will be executed, but if the argument of `new` is an array of type `private_complex`, then `pcarray_init` will be executed. The main program which uses this new module looks like the following:

```
      program main
! bring in complex_array (and private_complex) procedures
      use complex_array_class
      integer, parameter :: nx = 8
! define a single scalar of type private_complex
      type (private_complex) :: a
! define an array of type private_complex
      type (private_complex), dimension(nx) :: f
! initialize a single scalar
      call new(a,1.0,-1.0)
! initialize an array
      call new(f,(/(real(i),i=1,nx)/),(/(-real(i),i=1,nx)/))
      stop
      end program main
```

In a similar fashion we can add a multiply function to the new module which will multiply arrays. To do this we take advantage of a new feature of Fortran90, which is that functions can return entire arrays. This is done by declaring the function name to be an array, and setting its dimension from another array in the argument list with the SIZE intrinsic. Since the '`*`' operator for scalars of type `private_complex` has already been defined in the module private_complex_class, the array function is created by calling this operator in a loop, as follows:

```
      function pcarray_mult(this,b)
! multiply arrays of private_complex variables
      type (private_complex), dimension(:), intent(in) :: this, b
! declare output array to be of same size as "this" array
      type (private_complex), dimension(size(this)) ::pcarray_mult
      do i = 1, size(this)
! this multiplication is actually performed by function pc_mult
         pcarray_mult(i) = this(i)*b(i)
      enddo
      end function pcarray_mult
```

Finally, we can overload the '*' operator to call pcarray_mult when the arguments are arrays of type private_complex, as follows:

```
      interface operator(*)
         module procedure pcarray_mult
      end interface
```

In the final version of the derived class complex_array, which is listed in Appendix D, we have also overloaded the display function so that we can print out the array elements. In the following main program, we can now multiply arrays of type private_complex using array syntax:

```
      program main
      use complex_array_class
      integer, parameter :: nx = 8
! define sample arrays
      type (private_complex), dimension(nx) :: f, g, h
! initialize sample arrays
      call new(f,(/(real(i),i=1,nx)/),(/(-real(i),i=1,nx)/))
      call new(g,(/(-real(i),i=1,nx)/),(/(real(2*i),i=1,nx)/))
! perform multiplication of arrays
      h = f*g
! display the first three elements of the results
      call display(h(1:3),'h=')
      stop
      end program main
```

Thus we have constructed a derived class built upon the definitions and procedures in a base class using class composition. It was not necessary to construct a new derived type for arrays, since arrays of derived types are automatically supported in the language. Procedures which operate on the arrays, however, had to be constructed. This was done in two stages. First a new procedure for the derived class is written (for example, pcarray_mult) which executes the original base class operator ('*') in a loop. The original base class operator ('*') is then overloaded to include the new derived class procedure. Note that the base class was never modified during this process.

Inheritance is generally used to mean something more restricted than class composition. In this form of the inheritance relation, the base class contains the properties (procedures) which are common to a group of derived classes. Each derived class can modify or extend these procedures for its own needs if necessary.

As an example, consider the private_complex class discussed in the previous section. Suppose we want to extend this class so that it keeps track of the last operation performed. Such a feature could be useful in debugging, for example. Except for the additional feature of monitoring operations, we would like this extended class to behave exactly like the private_complex class. We can accomplish this by creating a new class called the monitor_complex class. First we create a new derived type, as follows:

```
type monitor_complex
   type (private_complex) :: pc
   character*8 :: last_op
end type monitor_complex
```

which contains one instance of a private_complex type plus an additional character component to be used for monitoring. We want to extend all three procedures of the private_complex class (new,'*',and display) so that they also work in the new monitor_complex class. We will accomplish this with methods very similar to those we used in composition. Initializing the monitor_complex type can be performed by making use of the new operator in the private_complex class to initialize the private_complex component of the monitor_complex type as follows:

```
type (monitor_complex) :: x
call new(x%pc,1.0,-1.0)
```

The additional monitor component can be initialized in the usual way:

```
x%last_op = 'INIT'
```

This initialization can be embedded in a procedure called mc_init. By also adding the interface new to the monitor_complex class, we can overload the new procedure so that it calls mc_init if the argument is of type monitor_complex. As a result, the operator new which previously worked on private_complex type has been extended to also work on monitor_complex types, as required. The resulting monitor_complex class looks like:

```
      module monitor_complex_class
! get (inherit) private_complex type and its public procedures
      use private_complex_class
! define monitor_complex type
      type monitor_complex
         private
         type (private_complex) :: pc
         character*8 :: last_op
      end type monitor_complex
      interface new
         module procedure mc_init
      end interface
      contains
         subroutine mc_init(this,real,imaginary)
! initialize monitor_complex variable from reals
         type (monitor_complex), intent(out) :: this
         real, intent(in) :: real, imaginary
! initialize private_complex component of monitor_complex
         call new(this%pc,real,imaginary)
         this%last_op = 'INIT'      ! set last operation
         end subroutine mc_init
      end module monitor_complex_class
```

In a similar fashion we can extend the multiplication function to also work on `monitor_complex` types. Multiplication is performed by using the '`*`' operator (defined in the `private_complex` class) on the `private_complex` component of the `monitor_complex` type, as follows:

```
         type (monitor_complex) :: x, y, z
         z%pc = x%pc*y%pc
```

This operation can be embedded in a function which returns a result of type `monitor_complex`. We also want to add to this function the operation of setting the monitor component. The resulting procedure can be written:

```
         type (monitor_complex) function mc_mult(this,b)
         type (monitor_complex), intent(in) :: this, b
         mc_mult%pc = this%pc*b%pc
         mc_mult%last_op = 'MULTIPLY'
         end function mc_mult
```

Finally, we can overload the '`*`' operator with the interface statement so that it calls `mc_mult` when the arguments are of type `monitor_complex`. In the final version of the derived class `monitor_complex`, which is listed in Appendix E, we have extended the display function to work with the `monitor_complex` class, as well as written an entirely new procedure (`last_op`) to display the last operation performed. In order to allow expressions with mixed types, we have also written a conversion operator `mc` to convert from `private_complex` to `monitor_complex` types. In the following main program, all of the procedures in the base class have been extended to work in the derived class:

```
      program main
```

```
! bring in monitor_complex definition and procedures
      use monitor_complex_class
! define sample variables
      type (private_complex) :: a, b, c
      type (monitor_complex) :: x, y, z
! initialize private_complex variables
      call new(a,1.,-1.)
      call new(b,-1.,2.)
! initialize monitor_complex variables
      call new(x,1.,-1.)
      call new(y,-1.,2.)
      call new(z,0.,0.)
! perform multiplication with private_complex types
      c = a*b
! perform multiplication with monitor_complex types
      z = x*y
! display results
      call display(c,'c=')
      call display(z,'z=')
! perform multiplication with mixed types
      z = mc(c)*z
! display last operation for monitor_complex type
      call last_op(z,'z')
      end program main
```

Thus we have constructed a derived class which has a special form of relationship to its base class. Here, the derived type definition in the derived class contains within it exactly one component of the derived type definition in the base class. In other words, each object of the derived class contains within it exactly one object of the base class. Furthermore all the procedures in the base class have been extended to also work in the derived class, although two have been internally modified, and one new one created. In addition, a conversion operator was supplied. Extending the procedures was accomplished by first writing a derived class procedure which called the base class procedure on the base class component, and then overloading the name to be the same as the base class procedure name. As in class composition, the base class was never modified during this process.

This form of inheritance is sometimes called subtyping because derived class objects can use base class procedures as if they were the same type. Although inheritance by subtyping is rather restrictive, it is also very convenient because special mechanisms exist in some languages (such as C++) to automatically extend unmodified base class procedures to derived types through automatic conversion of types. Fortran90 does not have such mechanisms and therefore inheritance by subtyping must be constructed explicitly, which can be cumbersome.

In many object-oriented languages, composition and subtyping are considered separately because they are implemented by two different language mechanisms, and composition is rarely considered a form of inheritance. However, in Fortran90, subtyping is implemented as a special case of composition and is constructed using similar methods, so that it makes sense here to consider the two ideas as related and describe them jointly. In our experience, composition is a more powerful idea which reflects more closely than subtyping how concepts are constructed in physics.

This example with `private_complex` types shows one of the reasons object-oriented programming is so powerful: if we decide to change the internal representation of `private_complex` to use polar coordinates instead of cartesian, we would have to modify only the procedures in the `private_complex` class to accommodate the change, while the derived classes would still work without modification. This example is perhaps academic. However, one can use the same techniques to create more powerful and interesting classes to represent other kinds of algebras. For example, one can create vector classes with procedures such as gradient operators, or tensor classes and associated procedures. One can then program at the same high level that one can do mathematics, with all the power and safety such abstractions give.

VI. Derived Types with Pointers

In the species_module created earlier (listed in Appendix B), we wrote a new subroutine push1 which had the following arguments:

```
subroutine push1(part,fx,species_args,dt)
```

The argument species_args was an instance of type species_descriptor that contained certain parameters describing a group of charged particles. The array part contained the particle coordinates for that group. This works well and the subroutine interface is considerably simplified from the original version.

Nevertheless, it is possible to add even more safety and simplicity. Clearly, the array part needs to have the object species_args always present. This leads to a possible source of error, since one can pass a descriptor which is inconsistent with the particle data. It makes sense therefore to make a new derived type which unites the two data types together. One might define such a type as follows:

```
type species
   type (species_descriptor) :: descriptor
   real, dimension(idimp,nop) :: coordinates
end type species
```

If idimp and nop are parameters known at compile time, such a definition is perfectly valid. However, it is not convenient, since changing idimp or nop would require much of the code to be recompiled. One might guess that it would be better to have an allocatable array in the type definition, such as:

```
type species
   type (species_descriptor) :: descriptor
   real, dimension(:,:), allocatable :: coordinates
end type species
```

It turns out that this is invalid: allocatable arrays cannot be used in derived type definitions. There is an alternative, however, which is valid, namely pointers to arrays. In order to explain how this works, we have to make a digression and explain how Fortran90 pointers work.

Pointers in any language refer to the location in memory of data, rather than the data itself. Fortran90 pointers are in reality a special kind of object. Compared to pointers in other languages, their use is greatly restricted in order to avoid the kind of performance degradation common with indiscriminant use of pointers. The programmer does not have access to the value of a pointer nor is pointer arithmetic permitted. Instead, pointers are really aliases to other data. Suppose we define two real arrays, and two pointers to real arrays:

```
real, dimension(4), target :: a, b
real, dimension(:), pointer :: ptr1, ptr2
```

One can then associate the first pointer with the array a using the '=>' operator as follows:

```
ptr1 => a
```

Notice that the TARGET attribute must be used for the arrays a and b, in order to allow them to be pointed to. The kind of data with which a pointer can be associated is determined in its declaration. Thus, the pointer ptr1 can only point to real, one dimensional arrays, and cannot point to a real, two dimensional array, for example. Once associated with a target, the pointer can then be used just as if it were the array a, including passing it as an argument to a procedure. This is called dereferencing. For example the statement

```
ptr1(1) = 1.0
```

will assign the value of 1.0 to a(1). The NULLIFY intrinsic is used to disassociate a pointer from an array:

```
nullify(ptr1)
```

Similarly, if we associate the second pointer with b,

```
ptr2 => b
```

then the statement

```
ptr2(2) = 2.0
```

will assign the value of 2.0 to b(2). The ASSOCIATED intrinsic function can be used to determine if a pointer has been associated with any array:

```
if (associated(ptr1)) print *,'ptr1 associated!'
```

It can also be used to determine if two pointers point to the same location in memory.

```
if (associated(ptr1,ptr2)) print *,'associated together!'
```

For our purposes, the most useful feature of pointers is that they can be associated with an unnamed variable with the ALLOCATE statement. For example, the statements

```
nullify(ptr1)
allocate(ptr1(4))
```

will first disassociate ptr1 from any previous array, then allocate an unnamed array consisting of 4 real words and associate the pointer ptr1 to that unnamed array. The pointer ptr1 can then be used as if it were a normal array. When we are finished with that array, we can deallocate it with:

```
deallocate(ptr1)
```

The ALLOCATED intrinsic does not work with pointers to arrays. However, the ASSOCIATED statement can tell us whether the data had actually been allocated, if we first NULLIFY ptr1 before allocating it. For all practical purposes, pointers to unnamed arrays function just like allocatable arrays, except that they have the advantage they can be used in derived type definitions.

Thus, the correct definition for the new species type is:

```
type species
   type (species_descriptor) :: descriptor
   real, dimension(:,:), pointer :: coordinates
end type species
```

If we define an object of type `species` called `electrons`,

```
type (species) :: electrons
```

then the pointer array for the particle data can be allocated as follows:

```
allocate(electrons%coordinates(idimp,nop),stat=ierr)
```

The new `push1` procedure can now be called with the simple statement:

```
call push1(electrons,fx,dt)
```

This can be organized efficiently by creating two classes. First, we create a `species_descriptor` class (see Appendix F) which contains the type definition for this class along with procedures to read and write objects of this class. This class is then "inherited" by a derived class called `species` (see Appendix G), which uses the base class information to define a `species` type along with a creation procedure and a new `push1` subroutine.

There is one subtle issue that occurs when using derived types containing pointers. When two such types are copied:

```
type (species) :: electrons, ions
ions = electrons
```

what actually occurs is

```
ions%descriptor = electrons%descriptor
ions%coordinates => electrons%coordinates
```

In the second line, the pointer component is copied, not the data to which it points. Sometimes this is not the desired behavior. If we want to copy the data instead of the pointer, we need to perform the following operation instead:

```
ions%coordinates = electrons%coordinates
```

which will dereference the pointer and copy the data. To accomplish this, we create a procedure:

```
subroutine copy_species(a,b)
type (species), intent(out) :: a
type (species), intent(in) :: b
a%descriptor = b%descriptor
a%coordinates = b%coordinates
end subroutine copy_species
```

Fortran90 allows such a procedure to be associated with the '=' operator by means of the `interface` statement in the module:

```
interface assignment (=)
   module procedure copy_species
end interface
```

If we implement such a procedure, then the statement:

```
ions = electrons
```

will copy the data instead of the pointer, assuming memory for the ion coordinates has been allocated.

VII. Dynamic Dispatching

One concept we have not discussed so far is the idea of dynamic dispatching, sometimes called run-time polymorphism, which is often said to be a distinguishing feature of object-oriented programming. The subtyping inheritance model we discussed in Section V was static, meaning that the compiler could resolve which procedure to call at any given point in the program. The purpose of dynamic dispatching is to allow one to write generic or abstract procedures which would work on all classes in an inheritance hierarchy, yet produce results that depend on which object was actually used at run-time. Dynamic dispatching is most useful when there are many objects which are similar to one another, but not identical, and which can be processed by some generic procedures. A common example occurs in database processing [8], where there are many similar kinds of records, students and teachers, for example. One wants to avoid writing different programs for each kind of record, yet one wants to handle each type of record differently.

To illustrate this, let us write a subroutine which does some kind of work using the methods in our `private_complex` class hierarchy. Since this inheritance hierarchy was rather simple, our `work` subroutine will also be simple: it will square a number and then print out the result:

```
      subroutine work(a)
      type (private_complex), intent(inout) :: a
      a = a*a
      call display(a,'work:')
      end subroutine work
```

We will define a `display` procedure so that it works differently in each class (for example, by modifying the `display` procedure in the `monitor_complex` class listed in Appendix E so that it calls `last_op`). The `work` subroutine is written for the `private_complex` type, but we would like it to function correctly even if we pass a `monitor_complex` type instead (which would normally be a type violation). In other words, when we say that this procedure works on a `private_complex` type, we actually mean that it is supposed to work on all types derived from `private_complex` as well. Furthermore, we want to decide at run time which we mean. Thus, if we declare and initialize the class objects as follows:

```
      type (private_complex), pointer :: pc
      type (private_complex), target :: a
      type (monitor_complex), target :: b
! initialize private_complex variables
      call new(a,1.,-1.)
! initialize monitor_complex variables
      call new(b,1.,-1.)
```

we would like to do something like:

```
! point to private_complex variable
      pc => a
      call work(pc)
! point to monitor_complex variable
      pc => b                        !error, type violation
      call work(pc)
```

This is not possible in Fortran90, because the pointer `pc` can only point to targets of type `private_complex`, so that the statement:

```
      pc => b
```

is illegal. The solution is to first define a derived type which contains a pointer to each of the possible types in the inheritance hierarchy, such as:

```
      type complex_subtype
         type (private_complex), pointer :: pc
         type (monitor_complex), pointer :: mc
      end type complex_subtype
```

This subtype has the ability to point to either complex type:

```
      type (complex_subtype) :: generic_pc
! point to private_complex variable
      generic_pc%pc => a
! point to monitor_complex variable
      generic_pc%mc => b
```

Dynamic dispatching can then be implemented by defining a `subtype` class which inherits all the classes in the inheritance hierarchy and contains a new version of each class member function. This new version will test which of the pointers in the complex_subtype type has been associated and execute the corresponding version of the function. For example, one can define a new display procedure as follows:

```
      subroutine display_subtype(a,c)
      type (complex_subtype), intent(in) :: a
      character*(*), intent(in) :: c
! check if pointer is associated with private_complex type
      if (associated(a%pc)) then
! if so, execute private_complex version of display
         call display(a%pc,c)
! check if pointer is associated with monitor-_complex type
      elseif (associated(a%mc)) then
! if so, execute monitor_complex version of display
         call display(a%mc,c)
      endif
      end subroutine display_subtype
```

The argument to this procedure is a variable of type `complex_subtype`. The procedure hides the decision about which actual function to call. In a similar fashion, one could write a new multiplication function, which would hide the decisions about the appropriate multiplication procedure to call. If one overloads the procedure names to have the same names as the corresponding class member functions, the resulting `work` procedure then has exactly the same form as before, except that the argument is of type `complex_subtype` rather than `private_complex`, to indicate that this subroutine is intended to be used with the entire class hierarchy:

```
      subroutine work(a)
      type (complex_subtype), intent(inout) :: a
! multiplication operator has been overloaded to cover all types
      a = a*a
      call display(a,'work:')
      end subroutine work
```

From the above, it is clear that the subtype class will need an assignment operator. Since we will make the components of the `complex_subtype` class `private`, one has to write a procedure to dynamically assign one of the possible pointers in the class hierarchy and `nullify` the rest. For example, the assignment of the `private_complex` pointer would be performed by:

```
      subroutine assign_pc(cs,pc)
      type (complex_subtype), intent(out) :: cs
      type (private_complex), target, intent(in) :: pc
! assign private_complex to complex_subtype
      cs%pc => pc
! nullify monitor_complex pointer
      nullify(cs%mc)
      end subroutine assign_pc
```

A similar procedure, which we call `assign_mc`, must be created to assign the `monitor_complex` pointer. Finally, in order for the following expression:

```
      a = a*a
```

to produce the expected results, one also needs to define a new copy operator, `copy_subtype`, which copies data rather than pointers, similar to the one we defined at the end of the section VI. These assignment and copy operators can be overloaded to the assignment operator (‘=’). The `complex_subtype` class which combines all these features is shown in Appendix H. It encapsulates all the details of how dynamic dispatching works, so that users of this class can freely write abstract or generic procedures based on the class hierarchy without being aware of how dynamic dispatching is implemented.

The program which calls `work` looks like:

```
program main
use complex_subtype_class
type (complex_subtype) :: pc
type (private_complex), target :: a
type (monitor_complex), target :: b
call new(a,1.,-1.)
call new(b,1.,-1.)
call assign(pc,a)
call work(pc)
call assign(pc,b)
call work(pc)
end program main
```

A distinguishing feature of typed object-oriented languages, is that support for the equivalent of such a subtype class is supported automatically by the compiler. There is always a performance penalty for using dynamic dispatching, however, even in object-oriented languages.

The rules for implementing a subtype class in Fortran90 which supports dynamic dispatching are as follows: create a derived type which contains exactly one pointer to each possible class in the inheritance hierarchy. Then implement a generic method for each class member function which will test which of the possible pointers have been associated and pass the corresponding pointer to the appropriate function. Assignment procedures are also required. The users of the subtype class, however, do not need to concern themselves with the details of how it is constructed.

Clearly, it is more cumbersome to implement dynamic dispatching in Fortran90 than in an object-oriented language. Once implemented, however, the usage is similar. Whether this feature is important depends on the nature of the problem one wants to model. Some studies of object-oriented programming [9] indicate that dynamic dispatching is needed about 15% of the time. There is some debate about what constitutes object-oriented programming. Some would argue that if a program does not use dynamic dispatching, it is not object-oriented. Others, however, argue that object-oriented is a question of design, not a question of what features of an object-oriented language are used. We tend to agree with the latter view.

VIII. Conclusions

Fortran90 is a modern, powerful language. There is much more here than array syntax, useful though that is. Many important programming concepts are supported, such as data encapsulation, function overloading and user defined types. The language also supports many safety features such as argument checking across procedures, implicit none, and intent attributes that enable the compiler to find many programming errors. Although Fortran90 is not considered an object-oriented language, a methodology can be developed to do object-oriented programming. The details of designing object-oriented programs are beyond the scope of this introductory article, but we have successfully written and compared Fortran90 and C++ versions of an object-oriented plasma particle-in-cell program [6-7]. But even if one did not wish to adopt the entire object-oriented paradigm, the programming concepts are very useful even if used selectively. It is possible to design simple, safer user interfaces to hide old, ugly procedural codes which may still work very well.

Acknowledgments:

The research of Viktor K. Decyk was carried out in part at UCLA and was sponsored by USDOE and NSF. It was also carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research of Charles D. Norton was supported by the NASA Graduate Student Researchers Program under grant NGT-70334, and that of Boleslaw K. Szymanski was partially supported by the NSF under grant CCR-9527151. We acknowledge the contribution of Steve Lantz, our friendly “in-house” reviewer, whose questions and comments improved the paper.

References:

[1] Michel Beaudouin-Lafon, Object-oriented Languages, translated by Jack Howlett [Chapman & Hall, New York, 1994].

[2] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, Object-Oriented Modeling and Design [Prentice-Hall, Englewood Cliffs, NJ, 1991]

[3] Kathleen Fisher and John C. Mitchell, “Notes on typed object-oriented programming,” in Theoretical Aspects of Computers Software, Proc. of International Symposium TACS ‘94, Sendai, Japan, April, 1994, ed. M. Hagiya and J. C. Mitchell [Springer-Verlag, Berlin, 1994], p. 844.

[4] T. M. R. Ellis, Ivor R. Philips, and Thomas M. Lahey, Fortran 90 Programming, [Addison-Wesley, Reading, Massachusetts, 1994].

[5] Stanley B. Lippman, C++ Primer, [Addison-Wesley, Reading, Massachusetts, 1991].

[6] Charles D. Norton, Boleslaw K. Szymanski, and Viktor K. Decyk, “Object-Oriented Parallel Computation for Plasma Simulation,” Communications of the ACM, vol. 38, no. 10, p. 88 (1995).

[7] Charles D. Norton, Viktor K. Decyk, and Boleslaw K. Szymanski, “On Parallel Object Oriented Programming in Fortran90,” ACM SIGAPP Applied Computing Review, vol. 4, no. 1, p. 27, 1996.

[8] R. Henderson and B. Zorn, “A Comparison of Object-Oriented Programming in Four Modern Languages,” Software-Practice and Experience, Vol. 24, Num. 11, pp. 1077-1095, Nov. 1994.

[9] Grady Booch, Object-Oriented Analysis and Design [Benjamin/Cummings, Redwood City, CA, 1994], p. 120.

Appendix A: Final version of fft1r_module

```
      module fft1r_module
      integer, save, private :: saved_indx = -1
      integer, dimension(:), allocatable, save, private :: mixup
      complex, dimension(:), allocatable, save, private :: sct
      contains
         subroutine fft1r_init(indx)
! initialization call
         integer, intent(in) :: indx
         integer :: nx, nxh, ierr, isign=0
! allocate f and t: old, ugly fft requires them as arguments
         real, dimension(2**indx) :: f
         complex, dimension(2**(indx-1)) :: t
         if (indx.lt.0) then
            print *,'indx must be non-negative'
            stop
         endif
         nx = 2**indx ; nxh = nx/2 ; saved_indx = indx
         if (allocated(mixup)) deallocate(mixup)
         if (allocated(sct)) deallocate(sct)
         allocate(mixup(nxh),sct(nxh),stat=ierr)
         if (ierr.ne.0) then
            print *,'allocation error'
            stop
         endif
! call old, ugly but fast fft here
         call lib_fft1r(f,t,isign,mixup,sct,saved_indx,nx,nxh)
         end subroutine fft1r_init
!
         subroutine fft1r_end
! deallocate internal data
         saved_indx = -1
         deallocate(mixup,sct)
         end subroutine fft1r_end
!
         subroutine fft1r(f,isign)
         real, dimension(:), intent(inout) :: f
         integer, intent(in) :: isign
         integer :: nx, nxh
         complex, dimension(size(f)/2) :: t
         nx = size(f) ; nxh = nx/2
! do nothing if isign is invalid
         if (isign.eq.0) return
         if (saved_indx.lt.0) then
            print *,'fft tables not initialized!'
            stop
         endif
! call old, ugly but fast fft here
         call lib_fft1r(f,t,isign,mixup,sct,saved_indx,nx,nxh)
         end subroutine fft1r
```

```
end module fft1r_module
```

Appendix B: Initial version of species_module

```
      module species_module
! define derived type
      type species_descriptor
         integer :: number_of_particles
         real :: charge, charge_to_mass, kinetic_energy
      end type species_descriptor
      contains
         subroutine push1(part,fx,species_args,dt)
! declare assumed-shape arrays
         real, dimension(:,:), intent(inout) :: part
         real, dimension(:), intent(in) :: fx
! declare argument of derived type
         type (species_descriptor), intent(inout) :: species_args
         real, intent(in) :: dt
         integer :: np, idimp, nop, nx
         real :: qbm, ek
! extract array sizes
         idimp = size(part,1) ; nop = size(part,2) ; nx = size(fx)
! unpack input scalars from derived type
         qbm = species_args%charge_to_mass
         np = species_args%number_of_particles
! call old, ugly but fast particle pusher here
         call orig_push1(part,fx,qbm,dt,ek,np,idimp,nop,nx)
! pack output scalars into derived type
         species_args%kinetic_energy = ek
         end subroutine push1
      end module species_module
```

Appendix C: Final version of private_complex class

```
      module private_complex_class
      private :: pc_init, pc_mult
      type private_complex
         private
         real :: real, imaginary
      end type private_complex
      interface new
         module procedure pc_init
      end interface
      interface operator(*)
         module procedure pc_mult
      end interface
      interface display
         module procedure pc_display
      end interface
      contains
         subroutine pc_init(this,real,imaginary)
! initialize private_complex variable from reals
         type (private_complex), intent(out) :: this
         real, intent(in) :: real, imaginary
         this%real = real
         this%imaginary = imaginary
         end subroutine pc_init
!
         type (private_complex) function pc_mult(this,b)
! multiply private_complex variables
         type (private_complex), intent(in) :: this, b
         pc_mult%real = this%real*b%real -
     &this%imaginary*b%imaginary
         pc_mult%imaginary = this%real*b%imaginary +
     &this%imaginary*b%real
         end function pc_mult
!
         subroutine pc_display(this,c)
! display value of private_complex variable with optional label
         type (private_complex), intent(in) :: this
         character*(*), intent(in), optional :: c
         if (present(c)) then
            print *, c, this%real, this%imaginary
         else
            write (6,'(2f14,7)',advance='no') this%real,
     &this%imaginary
         endif
         end subroutine pc_display
      end module private_complex_class
```

Appendix D: Final version of complex_array class

```
      module complex_array_class
      use private_complex_class
      private :: pcarray_init, pcarray_mult
      interface new
         module procedure pcarray_init
      end interface
      interface operator(*)
         module procedure pcarray_mult
      end interface
      interface display
         module procedure pcarray_display
      end interface
      contains
         subroutine pcarray_init(this,real,imaginary)
! initialize private_complex variable from reals
         type (private_complex), dimension(:), intent(out) :: this
         real, dimension (:), intent(in) :: real, imaginary
         do i = 1, size(this)
            call new(this(i),real(i),imaginary(i))
         enddo
         end subroutine pcarray_init
!
         function pcarray_mult(this,b)
! multiply arrays of private_complex variables
         type (private_complex), dimension(:),intent(in) :: this,b
         type (private_complex), dimension(size(this)):: 
     &pcarray_mult
! this multiplication is actually defined by function pc_mult
         do i = 1, size(this)
            pcarray_mult(i) = this(i)*b(i)
         enddo
         end function pcarray_mult
!
         subroutine pcarray_display(this,c)
! display value of private_complex array with label
         type (private_complex), dimension(:), intent(in) :: this
         character*(*), intent(in) :: c
         write (6,'(a)',advance='no') c
         do i = 1, size(this)
            call display(this(i))
         enddo
         print *
         end subroutine pcarray_display
      end module complex_array_class
```

Appendix E: Final version of monitor_complex class

```
      module monitor_complex_class
! get (inherit) private_complex type and its public procedures
      use private_complex_class
      private :: mc_init, mc_mult, mc_display
! define monitor_complex type
      type monitor_complex
         private
         type (private_complex) :: pc
         character*8 :: last_op
      end type monitor_complex
      interface new
         module procedure mc_init
      end interface
      interface operator(*)
         module procedure mc_mult
      end interface
      interface display
         module procedure mc_display
      end interface
      contains
         subroutine mc_init(this,real,imaginary)
! initialize monitor_complex variable from reals
         type (monitor_complex), intent(out) :: this
         real, intent(in) :: real, imaginary
! initialize private_complex component of monitor_complex
         call new(this%pc,real,imaginary)
! set last operation
         this%last_op = 'INIT'
         end subroutine mc_init
!
         type (monitor_complex) function mc_mult(this,b)
! multiply monitor_complex variables
         type (monitor_complex), intent(in) :: this, b
! this multiplication is actually defined by function pc_mult
         mc_mult%pc = this%pc*b%pc
! set last operation
         mc_mult%last_op = 'MULTIPLY'
         end function mc_mult
!
         subroutine mc_display(this,c)
! display value of monitor_complex variable with label
         type (monitor_complex), intent(in) :: this
         character*(*), intent(in), optional :: c
         call display(this%pc,c)
         end subroutine mc_display
!
```

```
         subroutine last_op(this,c)
! display last operation
         type (monitor_complex), intent(in) :: this
         character*(*), intent(in) :: c
            print *, 'last op for ', c, ' was ', this%last_op
         end subroutine last_op
!
         type (monitor_complex) function mc(pc)
! convert private_complex object to monitor_complex object
         type (private_complex), intent(in) :: pc
         mc%pc = pc
         mc%last_op = 'INIT'
         end function mc
      end module monitor_complex_class
```

Appendix F: Final version of species_descriptor class

```
      module species_descriptor_class
      type species_descriptor
         private
         integer :: number_of_particles
         real :: charge, charge_to_mass, kinetic_energy
      end type species_descriptor
      contains
         subroutine get_species(this,np,qm,qbm,ek)
! unpack components of species descriptor
         implicit none
         type (species_descriptor), intent(in) :: this
         integer, intent(out), optional :: np
         real, intent(out), optional :: qm, qbm, ek
         if (present(np)) np = this%number_of_particles
         if (present(qm)) qm = this%charge
         if (present(qbm)) qbm = this%charge_to_mass
         if (present(ek)) ek = this%kinetic_energy
         end subroutine get_species
!
         subroutine put_species(this,np,qm,qbm,ek)
! pack components of species descriptor
         implicit none
         type (species_descriptor), intent(out) :: this
         integer, intent(in), optional :: np
         real, intent(in), optional :: qm, qbm, ek
         if (present(np)) this%number_of_particles = np
         if (present(qm)) this%charge = qm
         if (present(qbm)) this%charge_to_mass = qbm
         if (present(ek)) this%kinetic_energy = ek
         end subroutine put_species
      end module species_descriptor_class
```

Appendix G: Final version of species class

```
      module species_class
! inherit species descriptor class
      use species_descriptor_class
      type species
         private
         type (species_descriptor) :: descriptor
         real, dimension(:,:), pointer :: coordinates
      end type species
      interface new
         module procedure species_init
      end interface
      contains
         subroutine species_init(this,species_args,idimp,nop)
! allocate particle coordinate pointer array and store descriptor
         type (species), intent(inout) :: this
         type (species_descriptor), intent(in) :: species_args
         integer, intent(in) :: idimp, nop
         integer :: ierr
! allocate pointer array
         allocate(this%coordinates(idimp,nop),stat=ierr)
! check for allocation error
         if (ierr.ne.0) then
            print *,'species allocation error'
! store descriptor
         else
            this%descriptor = species_args
         endif
         end subroutine species_init
!
         subroutine push1(this,fx,dt)
         type (species), intent(inout) :: this
         real, dimension (:), intent(in) :: fx
         real, intent(in) :: dt
         integer :: np, idimp, nop, nx
         real :: qbm, ek
! extract array sizes
         idimp = size(this%coordinates,1)
         nop = size(this%coordinates,2)
         nx = size(fx)
! unpack input scalars from derived type with inherited procedure
         call get_species(this%descriptor,np=np,qbm=qbm)
! call old, ugly but fast particle pusher here
         call orig_push1(this%coordinates,fx,qbm,dt,ek,np,idimp,
     &nop,nx)
! pack output scalar into derived type with inherited procedure
         call put_species(this%descriptor,ek=ek)
         end subroutine push1
      end module species_class
```

Appendix H: Final version of complex_subtype class

```
      module complex_subtype_class
! get (inherit) private_complex type and its public procedures
      use private_complex_class
! get (inherit) monitor_complex type and its public procedures
      use monitor_complex_class
      private
      public :: private_complex, monitor_complex, complex_subtype
      public :: new, assign, assignment(=), operator(*), display
! define complex_subtype type
      type complex_subtype
         private
         type (private_complex), pointer :: pc
         type (monitor_complex), pointer :: mc
      end type complex_subtype
      interface assign
         module procedure assign_pc
         module procedure assign_mc
      end interface
      interface assignment (=)
         module procedure copy_subtype
      end interface
      interface operator(*)
         module procedure mult_subtype
      end interface
      interface display
         module procedure display_subtype
      end interface
      contains
         subroutine assign_pc(cs,pc)
! assign private_complex to complex_subtype
         type (complex_subtype), intent(out) :: cs
         type (private_complex), target, intent(in) :: pc
         cs%pc => pc
         nullify(cs%mc)
         end subroutine assign_pc
!
         subroutine assign_mc(cs,mc)
! assign monitor_complex to complex_subtype
         type (complex_subtype), intent(out) :: cs
         type (monitor_complex), target, intent(in) :: mc
         nullify(cs%pc)
         cs%mc => mc
         end subroutine assign_mc
```

```
        subroutine copy_subtype(this,b)
! assign contents of complex_subtype to complex_subtype
        type (complex_subtype), intent(inout) :: this
        type (complex_subtype), intent(in) :: b
! check if pointer is associated with private_complex type
        if (associated(b%pc)) then
           this%pc = b%pc
           nullify(this%mc)
! check if pointer is associated with monitor_complex type
        elseif (associated(b%mc)) then
           this%mc = b%mc
           nullify(this%pc)
        endif
        end subroutine copy_subtype
!
        function mult_subtype(this,b) result(output)
! multiply complex_subtype variables
        type (complex_subtype), intent(in) :: this, b
        type (complex_subtype) :: output
        type (private_complex), target, save :: tpc
        type (monitor_complex), target, save :: tmc
! check if pointer is associated with private_complex type
        if (associated(this%pc)) then
           tpc = this%pc*b%pc
           output = tpc
! check if pointer is associated with monitor_complex type
        elseif (associated(this%mc)) then
           tmc = this%mc*b%mc
           output = tmc
        endif
        end function mult_subtype
!
        subroutine display_subtype(a,c)
! display value of complex_subtype variable with label
        type (complex_subtype), intent(in) :: a
        character*(*), intent(in) :: c
! check if pointer is associated with private_complex type
        if (associated(a%pc)) then
           call display(a%pc,c)
! check if pointer is associated with monitor_complex type
        elseif (associated(a%mc)) then
           call display(a%mc,c)
        endif
        end subroutine display_subtype
     end module complex_subtype_class
```