A Simplified Method for Implementing Run-Time Polymorphism in Fortran95

by

Viktor K. Decyk[1,2] and Charles D. Norton[1]

[1]Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109-8099

[2]Department of Physics and Astronomy
University of California, Los Angeles
Los Angeles, California 90095-1547

Abstract

This paper discusses a simplified technique for software emulation of inheritance and run-time polymorphism in Fortran95. This technique involves retaining the same type throughout an inheritance hierarchy, so that only functions which are modified in a derived class need to be implemented.

I. Introduction

Fortran 95 is modern language primarily used for high performance in scientific and technical computing. It is an object-based language, in our opinion, which directly supports many of the important concepts in object-oriented programming, including the creation of a hierarchy of objects, information hiding and encapsulation, and static polymorphism. It is not considered an object-oriented language, however, because it does not directly support the concepts of inheritance and run-time polymorphism. These concepts can be emulated, however, and in earlier papers [1-2], we showed how to do so. This allowed one to implement an object-oriented programming style in Fortran 95. The emulation of run-time polymorphism involved the creation of a special polymorphic class. This was straightforward to implement if the inheritance hierarchy was simple. However, it was cumbersome if the inheritance hierarchy was complex. In this paper, we show a simplified method for emulating run-time polymorphism. All of the methods discussed here also work equally well in the older standard, Fortran 90.

II. Classes

Why are such object-oriented concepts important for scientific programming? A class is defined to be a data type along with the functions that operate on that type. The main value of classes is to encapsulate and hide the complex details of a set of related operations while presenting a simplified set of functions for the programmer to use.

This encapsulation prevents inadvertent modification of internal data and also allows the implementation details to be changed without impacting the usage of the class. As a result, it is much easier for different programmers to implement different classes without getting in each others way, and it (usually) means that once a class is debugged, one does not have to worry about it further when debugging other new code. As a result, more complex computational projects can be attempted.

In our earlier paper, we used a stopwatch class as an example, first introduced by Gray and Roberts [3]. The class was intended to provide a simple means for timing parts of a program. It's usage is very simple. After first constructing a stopwatch, as follows:

```
type (stopwatch) :: sw
call new_stopwatch(sw)
```

One can then turn the stopwatch on and off in various parts of the program, such as:

```
call split(sw,'bar')       ! turn on stopwatch to time bar
call bar                   ! execute subroutine bar
call split(sw,'bar')       ! turn off stopwatch

call split(sw,'foo')       ! turn on stopwatch to time foo
call foo                   ! execute subroutine foo
call split(sw,foo')        ! turn off stopwatch
```

One can then report the results of these timings:

```
call report(sw,6)
```

where 6 refers to the output device. When the timings are done, delete the stopwatch:

```
call delete(sw)
```

Thus the class provides four functions (sometimes called methods) for the user, a constructor, a destructor, a split function and a report function. The details of what is inside the stopwatch type and how these functions are implemented do not need to be known by the user.

Of course, the person responsible for implementing this and related classes, needs to know these details. For example, one could define the class data members to be:

```
type stopwatch
   integer :: free
   character(len=mnl), pointer, dimension(:) :: name
   type (timer), pointer, dimension(:) :: split
end type stopwatch
```

The integer data member `free` is used to keep track of how many stopwatches are active at any time. The data member `name` is an array of character strings that contains

the name of each stopwatch, typically, the subroutine being timed. The data member `split` is an array of derived type elements that contains the current and elapsed time for each stopwatch.

III. Inheritance

Inheritance is used to create a hierarchy of related classes that have the same functions but different behavior. The child classes typically add data elements to the data of the parent's class and modify some but not all of the functions of the parent. Since the child class (also known as the derived class) always contains the data of the parent class (also known as the base class), it can execute the unmodified functions as if it were a parent. The type of the child class in such a case is also called a subtype of the base class. The main value of inheritance is to be able to create a set of related classes without duplicating existing code, while maintaining the same interfaces, that is, the same functions and arguments.

In an earlier paper [2], we created a parallel_stopwatch class which "inherited" from stopwatch. The parallel stopwatch was intended to provide additional functionality for timing parallel programs. In particular, it provided the global maximum time measured across the processors rather than the individual time on each processor. Since Fortran95 does not support inheritance as a language mechanism, we emulated the functionality of inheritance by first defining a new child type that contained within it, an object of the parent type. We added a new data item, `idproc`, to this type to keep track of the processor id for each parallel process. The new type was defined as follows:

```
type parallel_stopwatch
   private
   type (stopwatch) :: sw        ! base class component
   integer :: idproc             ! processor id
end type parallel_stopwatch
```

The inheritance relation requires that the child type provides the same functions as the parent, but the child could modify the functions, if desired. In this example, two of the four functions of stopwatch needed to be modified, the constructor and the report function. The new constructor is implemented by calling the constructor of the base class to construct the base class component sw, and a new procedure gidproc to obtain the processor id, as follows:

```
subroutine new_parallel_stopwatch(self,n)
type (parallel_stopwatch), intent(inout) :: self
integer, intent(in), optional :: n
call new_stopwatch(self%sw,n)     ! call base class constructor
call gidproc(self%idproc)         ! assign processor id
end subroutine new_parallel_stopwatch
```

The modified report function for the stopwatch class is implemented by first calling the normal stopwatch, then calling a parallel maximum function before reporting the results. For functions in the child class which are modified, the process here is very similar to the process used in object-oriented languages. In object-oriented languages,

however, unmodified functions defined in the base class automatically work in the derived class. This is not true in Fortran95, which does not allow one to use one type in place of another.

In this example, the other two functions in the parallel_stopwatch class, the destructor and the split method, did not need to be modified. In Fortran95, one needs to write a procedure with exactly one executable line which merely calls the base class procedure on the base class component of the derived type. For the split function, for example, one writes:

```
subroutine parallel_stopwatch_split(self,name)
type (parallel_stopwatch), intent(inout) :: self
character(len=*), intent(in) :: name
call split(self%sw,name)          ! delegate to base class
end subroutine parallel_stopwatch_split
```

In addition, one can overload the names of the functions so that they work with both types. For example, one overloads the name `split` as follows:

```
interface split
   module procedure parallel_stopwatch_split
end interface
```

After overloading the names, the usage of the parallel_stopwatch is exactly the same as the usage of the stopwatch shown earlier. Note that the child class did not have to reimplement any of the functions of the parent class. Except for the case where the child class inherits many unmodified functions, emulation of inheritance in Fortran95 is quite straightforward.

IV. Run-Time Polymorphism

Run-time polymorphism (also known as dynamic binding) allows a single object name to refer to any member of an inheritance hierarchy and permits a procedure to resolve at run-time which actual object is being referred to. This is useful because it allows one to write programs in terms of a single type which would behave differently depending on the actual type. Object-oriented languages support this behavior. Fortran95 does not, with the exception of elemental functions.

In order to emulate run-time polymorphism in Fortran95, two features must be constructed. The first feature is an identity mechanism to keep track of the actual class of an object in an inheritance hierarchy. The second is a dispatch mechanism (or method lookup) which selects the appropriate procedure to execute based on the actual class of the object.

In our earlier papers [1-2], we implemented the identity mechanism by defining a special polymorphic type consisting of pointers to all the possible types in the inheritance hierarchy, as follows:

```
type poly_stopwatch
  private
  type (stopwatch), pointer :: s
  type (parallel_stopwatch), pointer :: p
end type poly_stopwatch
```

At any given time, one of the pointers is associated with an actual object, the other pointers will be set to null objects. This polymorphic type functions like a base class pointer in C++. An assignment function was created to set the polymorphic type to one of the possible actual types. For example, to set the polymorphic type to a stopwatch type, one creates the function:

```
function assign_stopwatch(s) result(sw)
type (poly_stopwatch) :: sw
type (stopwatch), target, intent(in) :: s
sw%s => s         ! set poly_stopwatch to stopwatch component
nullify(sw%p)     ! nullify the parallel_stopwatch component
end function assign_stopwatch
```

To simplify usage, we overload the assign functions with the name `poly`, as follows:

```
interface poly
   module procedure assign_stopwatch
end interface
```

To implement the dispatch mechanism, one checks which of the possible pointers actually points to an object, then passes the associated pointer to the appropriate procedure. For example, the polymorphic split procedure would look like the following:

```
subroutine poly_stopwatch_split(self,name)
type (poly_stopwatch), intent(inout) :: self
character(len=*) name
if (associated(self%s)) then
   call split(self%s,name)
elseif (associated(self%p)) then
   call split(self%p,name)
endif
end subroutine poly_stopwatch_split
```

The following example illustrates how to use run-time polymorphism:

```
   use poly_stopwatch_class
   type (stopwatch) s             ! declare a stopwatch
   type (parallel_stopwatch) p    ! declare a parallel stopwatch
   type (poly_stopwatch) sw       ! declare a polymorphic stopwatch
!
   call new_stopwatch(s)           ! construct a stopwatch
   call new_parallel_stopwatch(p) ! construct a parallel stopwatch
!

! First use normal stopwatch
   sw = poly(s)             ! assign stopwatch to polymorphic type
   call split(sw,'bar')
   call bar()
   call split(sw,'bar')
   call report(sw,6)        ! report with normal stopwatch

! Then use parallel stopwatch
   sw = poly(p)             ! assign parallel stopwatch
   call split(sw,'foo')
   call foo()
   call split(sw,'foo')
   call report(sw,6)        ! report with parallel stopwatch
!
```

Implementing the polymorphic class is straightforward. One does not need to know the implementation details of these functions, just their argument types. In principle, the class can be implemented by a software tool. However, implementing it can be tedious if the hierarchy is deep. In particular, one must implement a dispatch mechanism for every function, even functions such as split, which are not modified by any class. Once created, however, the polymorphic class is used just like an object-oriented language such as C++ uses the base class pointer, and the user of the hierarchy does not need to know the implementation details.

V. Simplified Emulation of Run-Time Polymorphism

The reason emulating run-time polymorphism can be tedious is that Fortran95 is so strict about types, and our emulation was being faithful to the subtyping mechanisms used in object-oriented languages such as C++ or Java. Subtyping is easy in these languages, but not easy in Fortran95. If we abandon the unnatural use of subtyping in Fortran95, however, we discover that run-time polymorphism can be implemented much more simply, at the cost giving up some encapsulation.

Instead of creating a different type for each member of the inheritance hierarchy, and a special polymorphic type to implement dynamic dispatch, we will create a single expanded type for all of them, and use a different identity and dispatch mechanism to implement run-time polymorphism. This type includes the sum total of all the data members of every class in the inheritance hierarchy, as well as an extra flag, `subtype`,

which identifies the particular member of the inheritance hierarchy using this type. Thus for the stopwatch class hierarchy, we define the expanded type as follows:

```
type stopwatch
  integer :: subtype, free     ! subtype identifies actual type
  character(len=mnl), pointer, dimension(:) :: name
  type (timer), pointer, dimension(:) :: split
  integer :: idproc            ! used by parallel_stopwatch
end type stopwatch
```

Compared to the earlier definition of stopwatch, we have two extra members, `subtype` to identify the actual type, and `idproc`, which was used by the parallel_stopwatch class. This type must know about the data members of the entire hierarchy and must be modified whenever a new class in the inheritance hierarchy is created. It can function like a base-class pointer in C++. This type can be created in its own module, if desired.

The constructor for the stopwatch class requires an extra line, to set the value of subtype:

```
subroutine new_stopwatch(self,n)
...
self%subtype = 0       ! 0 identifies stopwatch class member
```

No further changes need to be made to the stopwatch class.

Implementing the inheritance hierarchy is simplified. The parallel_stopwatch class no longer has a separate type. Its constructor only needs to set the subtype component (to a different value):

```
subroutine new_parallel_stopwatch(self,n,timer_type)
...
self%subtype = 1  ! 1 identifies parallel_stopwatch class
```

The unmodified functions, destructor and the split, no longer have to be implemented as subroutines with one executable line. This is now more like inheritance in C++. The modified report function remains the same. However, the specific name `parallel_stopwatch_report` is used instead of the overloaded name "`report,`" since one no longer has different types to disambiguate the names. However, one can use the renaming facility in Fortran95 to control the names more exactly, if desired.

To implement polymorphism, we still create a polymorphic class module, but we no longer need a polymorphic type. A dispatch mechanism needs to be created only for the functions which were modified in the inheritance hierarchy. In this case, only the report function needs to have a dispatch mechanism. The `subtype` data member is used to select which actual function to call, as follows:

```
   subroutine poly_stopwatch_report(self,u)
   type (stopwatch), intent(inout) :: self
   integer :: u
   select case(self%subtype)
   case (0)                              ! normal stopwatch
      call stopwatch_report(self,u)
   case (1)                              ! parallel stopwatch
      call parallel_stopwatch_report(self,u)
   end select
   end subroutine poly_stopwatch_report
```

The other functions are either unmodified, or have unique names (the constructor). Since no separate type is used, no functions to assign the polymorphic types are necessary. Appendix I shows the entire inheritance hierarchy. Note that only functions which are actually modified or need to be polymorphic have to be implemented.

The following example illustrates how the simplified run-time polymorphism is used:

```
   use poly_stopwatch_class
   type (stopwatch), target :: s, p ! declare stopwatches
   type (stopwatch), pointer :: sw  ! declare a base class pointer
!
   call new_stopwatch(s)          ! construct a stopwatch
   call new_parallel_stopwatch(p) ! construct a parallel stopwatch
!
! First use normal stopwatch
   sw => s                        ! point to stopwatch
!
   call split(sw,'bar')           ! turn "bar" split on
   call bar()                     ! execute bar subroutine
   call split(sw,'bar')           ! turn "bar" split off
   call report(sw,6)              ! report total and split times

! Then use a parallel stopwatch
   sw => p                        ! point to parallel stopwatch
!
   call split(sw,'foo')           ! turn "foo" split on
   call foo()                     ! execute foo subroutine
   call split(sw,'foo')           ! turn "foo" split of
   call report(sw,6)              ! report total and split times
```

The fact that the type needs to know about the data members in the entire inheritance hierarchy is a violation of the usual encapsulation of data members in object-oriented languages. However, most Fortran programmers will find this a small price to pay for the simplicity of the implementation.

An approach similar to this involving geometric shapes was discussed by Stroustrup

[4]. He calls this approach a "mess" because "adding a new shape involves 'touching' the code of every important operation on shapes." This is not the case here, since we separate out the polymorphism from the individual implementations. The polymorphic functions only need to know about the interfaces of the individual subtypes, nothing about their implementation. Adding a new subtype involves only changing the contents of the type itself, perhaps a line or two, and adding two lines of code to each polymorphic function. Changing the type, however, will force a recompilation of the entire class. Fortran programs are rarely so long, however, that this is a serious problem.

VI. Application to Plasma Simulation

We have used this method of emulating run-time polymorphism in a Fortran95 based Framework designed to support parallel Particle-in-Cell (PIC) codes in plasma physics[5]. Such codes integrate self-consistently the trajectories of millions (now even billions) of particles in the electromagnetic fields created by the particles themselves. The basic structure of PIC codes[6] is to calculate a charge and/or current density from the particle co-ordinates, solve Maxwell's equations (or a subset) for the electric and/or magnetic fields, then advance the particles using Newton's law and the Lorentz force. The purpose of the Frameworks is to provide easy to use, trusted, components that are useful for building different kinds of PIC codes.

There are a variety of PIC codes in use. The most basic includes only the Coulomb electric field (solving only Poisson's equation). Others PIC codes include magnetic fields and solve either the full set of Maxwell's equation or the Darwin subset. We used our simplified method of modeling inheritance to implement a polymorphic particle type, as follows:

```
type species2d
  integer :: emf, ipbc      ! particle type, boundary condition
  real :: qm, qbm, dt       ! charge, charge/mass, time-step
  integer, dimension(:), pointer :: npp ! number of particles
  real, dimension(:,:), pointer :: part ! particle co-ordinates
end type species2d
```

The data member `emf` is used to distinguish different classes of particles, namely electrostatic, magnetized electrostatic, and electromagnetic. The base class is the electrostatic type, which includes only a charge deposit and a push subroutine which uses only electric forces. The magnetized electrostatic class modifies the push subroutine to include magnetic forces. The electromagnetic class adds a current deposit method to the hierarchy. Two polymorphic methods were added which used the `emf` data member to determine which push subroutine was used and whether the current deposit was actually called.

VII. Discussion and Conclusion

This new method of emulation of polymorphism is attractive because of its simplicity, but it has two shortcomings. First, it results in a type which includes information not needed by some classes. In particular, it can result in an object of a

base class containing some large unneeded data item from a derived class which is not used. To avoid this problem, we recommend that large data elements not be placed in such types, but rather pointers to such elements, which can be allocated when needed.

Another problem is that the data elements in a class are no longer private, and thus can be accessed and modified directly by any other class. In our own work, we never access the data elements directly outside of the class hierarchy where they are defined, so in fact, encapsulation is preserved. However, the language does not enforce this, as it does in object-oriented languages, so it is up to the programmer to preserve it. It is possible in principle to make the elements of the data type private and provide accessor functions for each data element but we doubt most Fortran programmers, who tend to be performance conscious, would tolerate the overhead. Finally, if enforcing such privacy is important, one can always revert back to our older method [1-2]. The older method is also useful when trying to add classes to a hierarchy whose internal details are not well understood or whose source code is unavailable. Programmers can use either method, depending on the situation.

Appendix I: Inheritance hierarchy for stopwatch classes

```
!
   module stopwatch_class
   use timer_class
   implicit none
!
   integer, parameter, private :: mnl=10
!
   type stopwatch
     integer :: subtype, free
     character(len=mnl), pointer, dimension(:) :: name
     type (timer), pointer, dimension(:) :: split
     integer :: idproc
   end type stopwatch
!
   interface delete
      module procedure delete_stopwatch
   end interface
!
   interface split
      module procedure stopwatch_split
   end interface
!
   contains
!
   subroutine new_stopwatch(self,n)
   type (stopwatch), intent(inout) :: self
   integer, intent(in), optional :: n
   integer i, max_splits
   self%subtype = 0
! make n names, splits
   if (present(n)) then
      max_splits = n
   else
      max_splits = 20
   endif
   allocate(self%name(0:max_splits))
   allocate(self%split(0:max_splits))
! blank all names, zero all splits
   do i = 0, max_splits
      self%name(i) = ' '
      call new_timer(self%split(i))
   enddo
! turn total timer on
   self%free = 0
   self%name(0) = 'total'
   call switch(self%split(0))
   end subroutine new_stopwatch
!
```

```
   subroutine delete_stopwatch(self)
   type (stopwatch), intent(inout) :: self
! deallocate split and name arrays
   deallocate(self%name,self%split)
   self%free = 0
   self%subtype = -1
   end subroutine delete_stopwatch
!
   subroutine stopwatch_split(self,name)
   type (stopwatch), intent(inout) :: self
   character*(*), intent(in) :: name
   integer i
   do i = 1, self%free
      if (self%name(i)==name) then
         call switch(self%split(i))
         return
      endif
   enddo
   if (self%free==size(self%name)) return
   self%free = self%free + 1
   self%name(self%free) = name
   call switch(self%split(self%free))
   end subroutine stopwatch_split
!
   subroutine stopwatch_report(self,u)
   type (stopwatch), intent(in) :: self
   integer, intent(in) :: u
   integer :: i
   real :: time, tot
   write (unit=u,fmt=*) '      TIMER STATISTICS (sec)'
   tot = real(timer_time(self%split(0)))
   do i = 1, self%free
      time = real(timer_time(self%split(i)))
      write (unit=u,fmt=*) self%name(i), time, '(', time/tot, '%)'
   enddo
   write (unit=u,fmt=*) self%name(0), tot
   write (unit=u,fmt=*) 'timer resolution = 1 /', resolution()
   end subroutine stopwatch_report
!
   end module stopwatch_class
!
```

```
!
   module parallel_stopwatch_class
   use stopwatch_class
   use parallel_class
   implicit none
!
   contains
!
   subroutine new_parallel_stopwatch(self,n)
   type (stopwatch), intent(inout) :: self
   integer, intent(in), optional :: n
   call new_stopwatch(self,n)        ! call base class constructor
   self%subtype = 1
   call gidproc(self%idproc)            ! assign processor id
   end subroutine new_parallel_stopwatch
!
   subroutine parallel_stopwatch_report(self,u)
   type (stopwatch), intent(in) :: self
   integer, intent(in) :: u
   integer :: i
   real, dimension(1) :: time, tot
   if (self%idproc==0) then
      write (unit=u,fmt=*) '      PARALLEL TIMER STATISTICS (sec)'
   endif
   tot(1) = real(timer_time(self%split(0)))
   call pmax(tot)
   do i = 1, self%free
      time(1) = real(timer_time(self%split(i)))
      call pmax(time)
      if (self%idproc.eq.0) then
         write (unit=u,fmt=*) self%name(i), time(1), '( ',       &
  &   time(1)/tot(1), ' %)'
      endif
   enddo
   if (self%idproc==0) then
      write (unit=u,fmt=*) self%name(0), tot
      write (unit=u,fmt=*) 'timer resolution = 1 /', resolution()
   endif
   end subroutine parallel_stopwatch_report
!
   end module parallel_stopwatch_class
!
```

```
!
   module poly_stopwatch_class
   use parallel_stopwatch_class
   implicit none
!
   interface report
      module procedure poly_stopwatch_report
   end interface
!
   contains
!
   subroutine poly_stopwatch_report(self,u)
   type (stopwatch), intent(inout) :: self
   integer :: u
   select case(self%subtype)
   case (0)
      call stopwatch_report(self,u)
   case (1)
      call parallel_stopwatch_report(self,u)
   end select
   end subroutine poly_stopwatch_report
!
   end module poly_stopwatch_class
!
```

Acknowledgments:

This research was 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 V. K. D. was also supported by the U. S. Department of Energy, under the SCIDAC program.

References:

[1] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to Express C++ Concepts in Fortran 90," Scientific Programming 6, No. 4, 1997, p. 363.

[2] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to Support Inheritance and Run-Time Polymorphism in Fortran 90," Computer Physics Communications 115, 9, 1998.

[3] M. G. Gray and R. M. Roberts, "Object-Based Programming in Fortran 90," Computers in Physics, 11, 355 (1997).

[4] Bjarne Stroustrup, The C++ Programming Language, [Addison-Wesley, Reading, MA, 1997], pp. 37-38

[5] C. Huang, V. Decyk, S. Wang, E.S.Dodd, C. Ren, W. B. Mori, "A Parallel Particle-in-Cell Code for Efficiently Modeling Plasma Wakefield Accleration: QuickPIC," Proc. of the 18th Annual Review of Progress in Applied Computational Electromagnetics, Monterey, CA, March, 2002, p.557.

[6] C. K. Birdsall and A. B. Langdon, Plasma Physics via Computer Simulation [Adam Hilger, New York, 1991].