

How to support inheritance and run-time polymorphism in Fortran 90

Viktor K. Decyk^{a,b}, Charles D. Norton^b, Boleslaw K. Szymanski^c

^a *Department of Physics and Astronomy, University of California, Los Angeles, Los Angeles, CA 90095-1547, USA*

^b *Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109-8099, USA*

^c *Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA*

Received 5 January 1998; revised 3 August 1998

Abstract

Fortran 90 does not support automatic inheritance and run-time polymorphism as language mechanisms. This paper discusses techniques for software emulation of inheritance and polymorphism in Fortran 90, which simplifies the implementation of an object-oriented programming style in Fortran 90. © 1998 Elsevier Science B.V.

1. Introduction

In recent years, a number of papers have appeared discussing Fortran 90 and Object-Oriented Programming (OOP) [1–6]. Fortran 90 clearly has some language features which are useful for OOP (derived types, modules, generic interfaces), but clearly lacks some others (inheritance, run-time polymorphism). Is OOP possible or practical in such a situation? Cary et al. [5] believe that Fortran 90 allows it “to some degree”, but that an object hierarchy cannot be constructed and polymorphic types are not possible. Gray and Roberts [6] argue that inheritance can be “faked” (sic), but the effort required too much duplication of code to be practical. In this paper we will show how to emulate inheritance and polymorphic types by software constructs without duplication of code. This allows one to implement all the important concepts in OOP, but with more effort than would be required in an object-oriented language. To keep the exposition of this paper clear and focused, we will expand on the stopwatch example used by Gray and Roberts.

What is inheritance? Gray and Roberts quote Rumbaugh [7] in defining inheritance to be “sharing of structure and behavior among classes in a hierarchical relationship”. They also define polymorphism to be “differentiation of behavior of the same operation on different classes”. In many object-oriented languages, inheritance and run-time polymorphism use the same language mechanism so these concepts are glued together. Since Fortran 90 does not have such a unifying mechanism, it is helpful to keep these concepts separate while developing emulation techniques. Another useful distinction is the difference between static (ad hoc) and run-time polymorphism. Static polymorphism means that the actual type being used at any point in the program is known at compile time, while run-time polymorphism means that a single type can refer to one of several possible actual types, and only at run-time can the correct type be determined.

When Gray and Roberts “fake” inheritance with their stopwatch example, they appear to conflate the concepts of inheritance and polymorphism together. This results in a situation where “developers of new classes derived from the stopwatch class must modify the base stopwatch class to accommodate the new type”, and where developers “must duplicate identical code in each of the child classes in order to allow future classes to override the default behavior”. Our techniques for implementing inheritance allow us to avoid these difficulties and replace the pejorative “fake” with the more neutral “emulate”.

2. Implementing an inheritance hierarchy

The real value in using inheritance is to avoid duplicating code when creating types (classes) which are similar to one another. Gray and Roberts create a base type called `stopwatch`, which is composed of a number of `timers` and has two main procedures: `split` (to toggle a timer on and off) and `report` (to display the results). They then create a class called `parallel_stopwatch`, which is derived from `stopwatch` and shares its procedures and interfaces, but adds new functionality for parallel computing. Unfortunately, they do not show how this derivation is done. Instead they focus on how to implement run-time polymorphism for these two classes. We will begin by showing how to create the derived class and later show how to implement run-time polymorphism.

We can define a parallel stopwatch class with the following derived type:

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

which contains exactly one instance of the base class type `stopwatch`, and an additional integer type which contains the processor id defined on a distributed memory parallel computer. A constructor for the derived class is implemented by calling the constructor of the base class to construct the base class component, and the procedure `gidproc` to obtain the processor id to initialize the integer component:

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

This is functionally equivalent to the memberwise initialization list used in C++ to initialize a derived class object. A similar procedure is used to create the destructor for the derived class. We want the method `split` in the derived class to work the same way as in the base class. In Fortran 90 this is implemented by writing a procedure with exactly one executable line which merely calls the base class procedure on the base class component of the derived type, as follows:

```
subroutine 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 stopwatch_split
```

In C++, such a procedure would be automatically available and would not have to be explicitly written.

Static polymorphism is implemented in Fortran 90 by using an interface block in the derived class, as follows:

```
interface split
  module procedure stopwatch_split
end interface
```

so that the name `split` can be used with either type correctly.

We want the method `report`, on the other hand, to be implemented differently in the derived class to accommodate the new features of parallel processing (e.g., it will report the maximum time returned by the stopwatches on each node). This method would have to be rewritten in both C++ and Fortran 90, and the two languages are equivalent here.

Thus we can create either a normal stopwatch or a parallel stopwatch and run one or the other as follows:

```
program main
! get stopwatch type definition and methods
  use parallel_stopwatch_class
#ifdef MPP
  type (parallel_stopwatch) sw ! declare a parallel stopwatch
#else
  type (stopwatch) sw ! declare a stopwatch
#endif
  call construct (sw) ! construct 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
  call destruct (sw) ! destroy stopwatch
```

Note that the `parallel_stopwatch` class did not have to “modify the base `stopwatch` class to accommodate the new type”, nor did we have to “duplicate identical code” in the child class.

So far, our emulation of inheritance consists of two techniques. First, inheritance of data members is implemented by including exactly one instance of the base class data member in a derived class. Second, inheritance of methods is implemented by delegation (or subcontracting) to the base class the responsibility of carrying out the operation on the base class component of the derived class object. These techniques for emulating inheritance have appeared earlier [3–5].

Inheritance is sometimes referred to as an “is-a” relation (a parallel stopwatch is a kind of stopwatch). Another relation which occurs in object-oriented programming is the “has-a” relation (a stopwatch has timers). Our emulation is based on the observation that a parallel stopwatch can function as a stopwatch precisely because it contains a stopwatch inside itself. This idea is not entirely new. Meyer [8] also points out that “when the “is” view is legitimate, one can always take the “has” view instead”.

What we have done is create an inheritance hierarchy but without run-time polymorphism (that is, without the use of virtual functions in C++). In many cases, this is sufficient. For example, by use of a preprocessor one can choose either a stopwatch or a parallel stopwatch at compile time. This is always more efficient at execution time than run-time polymorphism, even in C++. Another way to avoid requiring run-time polymorphism is to ensure that parallel stopwatches run correctly on serial computers. In our case this was implemented by requiring that the procedure which calculates a global maximum produces the correct result even if only a single node is being used. Then it is always safe to use parallel stopwatches even on non-parallel computers.

3. Implementation of run-time polymorphism

Nevertheless, sometimes one does desire the functionality of run-time polymorphism, and this is often considered a part of the meaning of inheritance. In the case of our stopwatches, the parallel stopwatch reports one global result, whereas the normal stopwatches report a result for each node separately. In normal operation, parallel stopwatches are used, but if unusual behavior is occurring, it may be desirable to switch to the normal stopwatch (possibly interactively) on the parallel computer to obtain information about the variation of timings across processors. We implement run-time polymorphism by creating a polymorphic type called `poly_stopwatch`, which contains a pointer for each possible type in the inheritance hierarchy:

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

If stopwatches have already been created, then one can write a conversion function to assign one of the pointers in the polymorphic type to the stopwatch one desires to use (nullifying the others) as follows:

```
function convert_stopwatch(s) result(sw)
! convert stopwatch to poly_stopwatch
type (poly_stopwatch) :: sw
type (stopwatch), target, intent(in) :: s
sw%s => s
nullify(sw%p)
end function convert_stopwatch
```

If we further write a similar conversion function for parallel stopwatches and create the interface name `poly` to refer to them:

```
interface poly
  module procedure convert_stopwatch
  module procedure convert_parallel_stopwatch
end interface
```

then we can use the `poly_stopwatch` type to refer to either type:

```
type (stopwatch) s           ! declare a stopwatch
type (parallel_stopwatch) p ! declare a parallel stopwatch
type (poly_stopwatch) sw    ! declare polymorphic stopwatch
!
call construct(s)           ! construct a stopwatch
call construct(p)          ! construct a parallel stopwatch
!
sw=poly(s)                  ! sw is a normal stopwatch now
....
sw=poly(p)                  ! sw is a parallel stopwatch now
....
```

In addition to creating a polymorphic type, a dispatch mechanism must be constructed to execute the correct procedure. This procedure merely checks which pointer has been associated, and executes the corresponding procedure.

```

subroutine stopwatch_report (self,u)
type (poly_stopwatch), intent (inout) :: self
integer, intent(in) :: u
if (associated(self%s)) then
  call report(self%s,u)
elseif (associated(self%p)) then
  call report(self%p,u)
endif
end subroutine stopwatch_report

```

which is similar to the one shown by Gray and Roberts. When this function is added to the interface report, it corresponds to creating a virtual function in C++. The following example illustrates how to use run-time polymorphism:

```

sw=poly(s)           ! use normal 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
!
sw=poly(p)           ! use parallel stopwatch
call split(sw,'foo') ! turn "foo" split on
call foo()           ! execute foo subroutine
call split(sw,'foo') ! turn "foo" split off
call report(sw,6)    ! report total and split times

```

This use of the `poly` function with the `poly_stopwatch` type corresponds to the assignment of derived class objects to base class pointers in C++.

This polymorphic class functions like an abstract base class or interface class in C++. However, it is constructed after all the classes in the inheritance hierarchy are known, and it is the only place where such knowledge is concentrated. Such a situation corresponds to a polymorphic instance set which can occur in object-oriented programming as described by Meyer [8]. The polymorphic class knows only about the types and interfaces in the hierarchy and nothing whatsoever about their implementation. None of the classes in the hierarchy that we have created with static polymorphism have to be modified to implement run-time polymorphism. Therefore, it should be possible to write a software tool to automatically create such a polymorphic class. This would be a useful project for some enterprising computer science student!

One interesting feature of this approach to adding run-time polymorphism in Fortran 90 is that the polymorphic class can consist of any types whatsoever, not necessarily those related by inheritance as in C++. Although this is not exactly the same as templates in C++, it serves a similar purpose: the ability to write one function which can be used with different actual types. This answers the concern of Cary et al. that “there is no way to refer to a group of ... objects collectively and have [functions] return what is appropriate for each object”.

If a new type of stopwatch is added to the inheritance hierarchy, derived from one of the other stopwatches, one first implements an inheritance relationship with static polymorphism, which does not require changing any of the previous classes, as we have shown. If one further desires to add run-time polymorphism to this third class, an additional pointer must be added to the polymorphic type, an additional line or two must be added to the conversion functions and methods. Finally, a new conversion function must be created. In this example, this corresponds to adding about a dozen lines of new code.

As an alternative to (or in addition to) creating conversion functions, which requires one to first create a specific stopwatch, then assign it to the polymorphic type, one can create a constructor which internally creates

one of the stopwatches and associates the corresponding pointer in the polymorphic type, as follows:

```

integer, save, private :: platform=PARALLEL
!
subroutine stopwatch_construct(self,n,timer_type)
type (poly_stopwatch), intent(out) :: self
integer, intent(in), optional :: n, timer_type
if (present(timer_type)) platform=timer_type
if (platform==PARALLEL) then
  allocate(self%p)
  call construct(self%p,n)
  nullify(self%s)
else
  allocate(self%s)
  call construct(self%s,n)
  nullify(self%p)
endif
end subroutine stopwatch_construct

```

This is similar to the constructor shown by Gray and Roberts, and the following example illustrates its use:

```

program main
use poly_stopwatch_class
type (poly_stopwatch) sw      ! declare a polymorphic type
!
call construct(sw)           ! construct 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
call destruct(sw)           ! destroy stopwatch

```

4. Discussion

Except for the discussion of inheritance, Gray and Roberts have an excellent discussion of how to model object-oriented concepts in Fortran 90. There are only a small number of other improvements which we can suggest to their exposition. One minor point is that the `implicit none` statement does not have to be repeated in each subroutine. It can be declared just once in each module and will automatically apply to each procedure contained in that module. Another minor point is that it is not necessary to make a complete list of the `public` and `private` entities in a module separately. One can make the default `public` or `private` and then list only the exceptions. Another improvement which one can take advantage of is that pointers in Fortran 90 are more “intelligent” than pointers in C++, because they know how much memory has been allocated to them. Therefore, class data members such as `max_splits` in the `stopwatch` class described by Gray and Roberts are not needed, since one can always obtain the size from the Fortran 90 `size` intrinsic, for example:

```

type(stopwatch) :: sw
allocate(sw%name(20))
max_splits=size(sw%name)

```

A further useful feature in Fortran 90 is the idea of an optional argument. Gray and Roberts implement two constructors for the base stopwatch class, `stopwatch_construct` and `stopwatch_construct_1`. The only difference between them is that in the former case the number of `timers` defaults to 20, and in the latter case it is explicitly given as an argument. Two constructors are not required if one uses optional arguments, as follows:

```

subroutine stopwatch_construct(self,n)
type (stopwatch), intent(out) :: self
integer, intent(in), optional :: n
integer i, max_splits
! make n names, splits
if (present(n)) then
  max_splits=n
else
  max_splits=20
endif
....

```

These optional arguments are more powerful than default arguments in C++, because they can be referenced by keyword. For example, the constructor for the `poly_stopwatch` type previously discussed above,

```

subroutine stopwatch_construct(self,n,timer_type)
type (poly_stopwatch), intent(out) :: self
integer, intent(in), optional :: n, timer_type
if (present(timer_type)) platform=timer_type
....

```

can be called with the second optional argument but not the first as follows:

```

type (poly_stopwatch) sw          ! declare a polymorphic type
call construct (sw,timer_type=0) ! construct SERIAL stopwatch

```

Gray and Roberts use the term “object-based programming” because their emulation of inheritance “required far too much work for far too little benefit” to use. We feel that our techniques make object-oriented programming more practical in Fortran 90. Indeed, we have translated a great many object-oriented examples from C++ to Fortran 90, and have not yet found any object-oriented concept as defined by Gray and Roberts which could not be implemented. Many of these examples are available on our web site [9], including a complete listing of our implementation of stopwatches. For an extended discussion of the use of polymorphic types in Fortran 90, see Ref. [3].

Cary et al. give a good explanation of inheritance by delegation, but they are clearly less proficient in Fortran 90 than in C++, and as a result their paper has a number of errors, misunderstandings, and inefficiencies. Many of the errors would be caught by a compiler, so we will not dwell on them here. However, there are some points that are more subtle that we feel should be explicitly addressed. In their discussion, they have a procedure to return the kinetic energy of a particle, similar to

```

real function KineticEnergy(p)
type (particle), intent(in) :: p
real :: ke=0.0
ke=ke+(p%velocity)**2
KineticEnergy=p%mass*ke
end function KineticEnergy

```

The problem here is the initialization of the variable `ke`. In Fortran 90, when a variable is declared with an

initial value in a procedure, it automatically has the **save** attribute, so that on second entry, the old value of **ke** would have been used, instead of being reinitialized to zero. This is a common trap when translating C++ code to Fortran 90.

A misunderstanding they have involves the **parameter** attribute, as in the following declaration:

```
real, parameter, private :: elemCharge=1.6e-19
save
```

The author state that “The SAVE qualifier indicates that only one copy of the parameter . . . is used”. Parameters are not variables, they are symbolic constants. There is no danger of having multiple constants, but the **save** statement is harmless. Other minor improvements one could make to their exposition is that **return** statements are not needed at the end of a procedure, they are automatic. Also, they do not take advantage of array syntax, but continue to write loops in the C++ style.

There are three main disadvantages to using object-oriented techniques in Fortran 90 compared to doing so in a true object-oriented language. The first is that more code must be written, especially in implementing the polymorphic class. This situation could be improved by the development of an appropriate software tool. The second is that emulation of inheritance requires an extra layer (or more) of procedure calls. This can lead to performance degradation if the amount of work being done in the procedure is small. Finally, procedures which use polymorphic types must be recompiled if a new derived class is added to the inheritance hierarchy. This can be a nuisance for very large projects where compilation time is long.

We have been programming in Fortran 90 and C++ for several years now [10,11], and have found that there are indeed many useful ideas in object-oriented analysis and design. The two concepts which we have found most useful are the notion of creating simple, stable interfaces for procedures by data hiding and encapsulation with types and the idea of avoiding replication of code.

Other ideas seem less compelling. Gray and Roberts argue that “inheritance is . . . the most important concept in science. Knowing that an emu is a kind of bird tells me many things about its behavior”. But inheritance can also be misused. For example, by inheritance one might conclude that emus fly. (Oops, they don’t!)

The inheritance relationship as traditionally defined in object-oriented languages requires that exactly one copy of the base class data members are contained inside a derived class. The relationship is like that of the Russian matryoshka dolls, where one fits snugly inside another. We have not found very many examples where such a relationship occurs in our scientific programming, although we recognize that such examples exist in other domains. Instead, the notion of composition, where one or more objects are contained inside another (such as timers in stopwatches) occurs more often. The notion of delegation which we used to model inheritance can also be used here to avoid replicating code, just as in C++. Even Rumbaugh [7] notes that “Many applications do not require inheritance. Many other applications have only a few classes requiring inheritance”.

The few times when run-time polymorphism seemed useful often occurred for families of data types which were not related by inheritance, while object-oriented languages supported polymorphism only when the data was related by inheritance. Object-oriented programming seems to be based on the idea that types are more important than procedures. We are still not convinced this is true, nor are we convinced that the opposite is true. But we do have some nagging doubts whether nouns are more central than verbs in the scheme of things.

Fortran 90 is a language designed for scientific computing and we believe it does this well. C++ is designed as a more general purpose language which deliberately avoids specialization to specific domains. Each language has both advantages and disadvantages. Fortran 90 has specialized features useful for scientific calculations, such as powerful array classes, but requires one to write polymorphic classes. C++ has a powerful inheritance mechanism, but requires one to write array classes. We have found that it takes significantly longer to debug code written in C++ than in Fortran 90. One reason is that Fortran 90 is more restricted in its features and more strict in its type checking (no automatic conversions across arguments, for example). Another reason has to do with style. Meyer describes a “picture of the software developer as fireworks expert or arsonist. He prepares a giant conflagration . . . then lights up a match and watches the blaze”. Such code can be very difficult to debug

when some unexpected problem arises, especially in C++ when polymorphism and automatic type conversions obscure the flow of logic. It makes one long for the simple days of spaghetti code, when at least there were statement labels to help one find ones way through all that pasta!

Acknowledgements

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 a National Research Council Associateship, and that of Boleslaw K. Szymanski was sponsored under grants CCR-9216053 and CCR-9527151.

References

- [1] B.J. Dupee, Object oriented methods using Fortran 90, ACM Fortran Forum 13(1) (1994) 21.
- [2] L. Machiels, M.O. Deville, Fortran 90: an entry to object-oriented programming for the solution of partial differential equations, ACM Trans. Math. Software 23 (1997) 32.
- [3] V.K. Decyk, C.D. Norton, B.K. Szymanski, How to express C++ concepts in Fortran 90, Scientific Programming (1998), to be published.
- [4] V.K. Decyk, C.D. Norton, B.K. Szymanski, Expressing object-oriented concepts in Fortran 90, ACM Fortran Forum 16 (1997) 13.
- [5] J.R. Cary, S.G. Shasharina, J.C. Cummings, J.V.W. Reynders, P.J. Hinker, Comparison of C++ and Fortran 90 for object-oriented scientific programming, Comput. Phys. Commun. 105 (1997) 20.
- [6] Mark G. Gray, Randy M. Roberts, Object-based programming in Fortran 90, Comput. Phys. 11 (1997) 355.
- [7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modelling and Design (Prentice Hall, Englewood Cliffs, NJ, 1991).
- [8] B. Meyer, Object-Oriented Software Construction (Prentice Hall, Upper Saddle River, NJ, 1997).
- [9] <http://www.cs.rpi.edu/~szymansk/oof90.html>
- [10] C.D. Norton, B.K. Szymanski, V.K. Decyk, Object-oriented parallel computation for plasma simulation, Comm. ACM 38(10) (1995) 88.
- [11] C.D. Norton, V.K. Decyk, B.K. Szymanski, High performance object-oriented scientific programming in Fortran 90, Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Computing, Minneapolis, MN, March, 1997, M. Heath et al., eds. (SIAM, Philadelphia, PA, 1997), CD-ROM.