Expressing Object-Oriented Concepts in 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

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109-8099

email: nortonc@olympic.jpl.nasa.gov

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

Abstract

Fortran90 is a modern, powerful language with features that support important new programming concepts, including those used in object-oriented programming. This paper briefly summarizes how to express the concepts of data encapsulation, function overloading, classes, objects, inheritance, and dynamic dispatching.

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, including some of those used in object-oriented programming, but scientific programmers generally are aware of only one of them: array syntax. In this paper, we will summarize the concepts of data encapsulation, function overloading, classes and objects, inheritance, and dynamic dispatching and explain how they can be expressed in Fortran90. Specific details can be found in our other publications [1-3].

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 programs typically 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 a migration path to parallel computers, since High Performance Fortran (HPF) is also based on Fortran90.

II. Data Encapsulation, Information Hiding, and Function Overloading

A fundamental idea behind object-oriented programming is that users of one program unit should know as little as possible about what is inside other program units that they use. This idea is called information hiding. Not only does it make program units easier to use, but it allows internal details of program units to change without impacting the users. A related idea is data encapsulation, which means that data which is needed in only one program unit will not be accessible and cannot be changed by another. This leads to enhanced program safety. Although such techniques were possible in Fortran77, they were error prone. Fortran90 has features to make it much easier and safer. As an example, consider a call to an FFT library routine in Fortran77:

```
      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, while `nxh` (`=nx/2`) is the size of the remaining arrays. The variable `isign` determines the direction of the transform.

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. Such encapsulation and information hiding can be achieved by means of automatic, allocatable, and assumed-shape arrays. Since `t` is a scratch array needed only during the call, it is best treated as an automatic array. The tables needed by the FFT should remain between calls, so they are best treated as saved, allocatable arrays. The input array `f` is best treated as an assumed-shape array because such arrays know their own size.

With such encapsulation, the author of the FFT can make changes to the algorithm or the internal data structure and the user of the subroutine would not be impacted. Another benefit is that one can hide old Fortran77 procedures behind a modern Fortran90 interface. For example, the Fortran90 FFT can actually call the original Fortran77 FFT inside. Then, the original FFT can be replaced (perhaps with a more optimized version) without the users of the FFT having to modify anything. With these language features, it is easier to develop software collaboratively, since individual authors responsible for individual procedures can make internal changes without affecting the other collaborators.

One powerful new feature of Fortran90 is the ability to check whether the number of types of arguments to called procedures are consistent with their declarations. This means that if one accidentally called the FFT procedure as follows:

```
call fft1r(f)
```

the compiler would complain that the argument `isign` was missing. Argument checking is automatically provided for functions defined in a new program unit called a module.

This ability to check arguments allows function overloading, which refers to using the same function name or operator symbol but performing different operations based on argument type. In Fortran77, this has always been available for intrinsic operations. For example, the '/' symbol gives different results depending on whether its arguments are real, integer, or complex. Fortran90 extends this ability to user defined functions and operators by means of the generic function mechanism. Note that function overloading is done at compile time and does not incur any performance penalty during execution.

For example, if we have another FFT procedure called `fft2r` which works on 2 dimensional data:

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

then both FFT procedures can be given the same name `fft` (which is now overloaded) because the arguments `f` are of different types: in one case `f` is a 1d array, and in the other it is a 2d array. If the both procedures are in one module, this overloading is done with the following declaration:

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

III. Derived Types, Classes, and Objects

Fortran90 allows users to define their own data types, built from intrinsic types such as real and integer, as well as other previously defined types. These are known as abstract data types or derived types. Derived types are structures or records (common in other languages) which can store items of possibly different types together. For example, one can define a `private_complex` type to consist of two real components as follows:

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

To create variables `a`, `b`, and `c` of this new type, one makes the following declaration:

```
type (private_complex) :: a, b, c
```

The components of this new type are accessed with the '`%`' symbol, and one can assign values as follows:

```
a%real = 1.0
a%imaginary = 2.0
```

If we write a procedure for multiplication of the `private_complex` types, it makes sense to place the new derived type together with the procedures which operate on that type into the same module, as follows:

```
      module private_complex_module
! define private_complex type
      type private_complex
         private
         real :: real, imaginary
      end type private_complex
```

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

What we have created here is a simple class. It consists of a single derived type definition (whose components are the class data members) along with the procedures which operate only on that type (called the class member functions or methods). The actual variable of type `private_complex` is called the object. Multiplication of two objects is performed by calling the procedure:

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

In this example we have made the components of `private_complex` type PRIVATE so that they are directly accessible only to the class member functions. In other words, the object `a` is available in the main program, but the components `a%real` and `a%imaginary` are not. Then any changes made to the internal representation of the `private_complex` type (for example, switching to polar coordinates from Cartesian) would be confined to this module and would not impact program units in other modules (assuming the public subroutine arguments remain the same).

The example of `private_complex` is perhaps academic, since the complex type already exists in Fortran. However, similar techniques can be used to create more powerful and interesting classes to represent others kinds of algebras and thereby program at the same high level with which one can do mathematics, with all the power and safety such abstractions give.

IV. Inheritance

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. Generally, the base class contains the properties (meaning data and procedures) which are common to a group of derived classes. Each derived class can then modify or extend each of these for its own needs if necessary.

As an example, suppose we want to extend the `private_complex` 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` that "uses" the types and procedures defined in the `private_complex_class`. In this new class, we define 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. Then we extend all the procedures defined in the `private_complex` class so that they work in the `monitor_complex` class. For multiplication, we create a `mc_mult` procedure which calls `pc_mult` on the `private_complex` part of `monitor_complex` as follows:

```
module monitor_complex_class
use private_complex_class
type monitor_complex
   private
   type (private_complex) :: pc
   character*8 :: last_op
end type monitor_complex
contains
   function mc_mult(a,b) result(c)
   type (monitor_complex), intent(in) :: a, b
   type (monitor_complex) :: c
   c%pc = pc_mult(a%pc,b%pc)
   c%last_op = 'MULTIPLY'
   end function mc_mult
end module monitor_complex_class
```

If we overload the '*' operator to refer to `pc_mult` and `mc_mult` with the INTERFACE statement:

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

then the multiplication of each type looks the same

```
type (private_complex) :: a, b, c
type (monitor_complex) :: x, y, z
c = a*b    ! multiplication with private_complex types
z = x*y    ! multiplication with monitor_complex types
```

but multiplication in the derived `monitor_complex` behaves differently than multiplication in the base class `private_complex` because it stores the `last_op` component.

V. Dynamic Dispatching

The purpose of dynamic dispatching (or run-time polymorphism) is to allow one to write generic or abstract procedures which work on all classes in an inheritance hierarchy, yet produce results that depend on which object was actually used at run-time. To illustrate this, suppose we had written a subroutine called `work` which made use of the functions in the `private_complex` class hierarchy to 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
```

where some appropriate `display` procedure had been defined. Since object `a` has been declared of type `private_complex`, this `work` subroutine will not function if a `monitor_complex` type was passed, even though multiplication and display are defined for that type. One could, of course, write another `work` procedure which is identical to this one except that object `a` is declared of type `monitor_complex`. This can be tedious and error prone, however. Dynamic dispatching allows one to write a single procedure that would work with both types. In object-oriented languages, such capabilities are normally available automatically. In Fortran90, one must write a special subtype class to provide this functionality. Details of how this class is constructed are beyond the scope of this article, but the interested reader can refer to our other publications.

References:

[1] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "Introduction to Object-Oriented Concepts using Fortran90," submitted for publication.

[2] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to Express C++ Concepts in Fortran90," submitted for publication.

[3] See also the web site: http://www.cs.rpi.edu/~szymansk/oof90.html

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