Do we need full object-oriented features?

Mike Fisher

ECMWF

18 November 2009

Outline

ECMWF

Outline

ECMWF

Definition of "Object-Oriented"

Object-oriented programming deals with objects.

An object is a combination of data, and functions that act on that data.

In general, an object restricts access to its data. Its functions provide an interface to other objects.

The functions are regarded as being part of the object, and are carried round with it.

Object-oriented languages regard programs as collections of interacting objects.

In contrast, procedural languages (such as Fortran 90/95) regard data and functions (subroutines) as distinct. The emphasis is on the functions rather than on the data. Functions are global entities, accessible from anywhere, and not attached to particular data structures

Definition of "Object-Oriented"

The primary mechanism used by object-oriented languages to define and manipulate objects is the class

Classes define the properties of objects, including:

- The structure of their contents,
- The visibility of these contents from outside the object,
- The interface between the object and the outside world,
- What happens when objects are created and destroyed.

There is no direct equivalent of the class in Fortran 90/95, although several properties of classes can be simulated using modules.

Definition of "Object-Oriented"

Different definitions of "object oriented" stress different aspects. But most agree that the following concepts are important:

- **Encapsulation**
- **Inheritance**
- **Polymorphism**
- **Abstraction**

The rest of this talk will consider each of these concepts in turn.

Let's start by considering encapsulation...

ECMWF

Outline

ECMWF

Encapsulation

In c++:

```
class Animal {
private:
  double health=50, happiness=50, hunger=50;
public:
  void eat()   {hunger=0.5*hunger;}
  void play() {happiness=50+0.5*happiness;}
  virtual void makeNoise () {
    cout << "woofoink" << endl;
  }
};
```

With this definition, we can make and manipulate objects of class Animal:

```
Animal lassie, babe;
lassie.eat();
babe.play();
lassie.makeNoise();
```

ECMWF

Encapsulation

Note how the class defines:

- the contents of Animal objects: health, happiness, etc.,
- the visibility of the contents: private,
- the interface by which the contents are accessed: eat, play, etc.

Note also that the important information describing an Animal is contained within and manipulated by the Animal. It is not held in some separate common block. It is not initialised in a separate set-up routine.

This is encapsulation.

We can do much the same using modules in Fortran 90/95:

```
module Animal_module
  type :: Animal_type
    private
    real :: health=50, happiness=50, hunger=50
  end type Animal_type
contains
  subroutine eat(this)
    type(Animal_type) :: this
    this%hunger = 0.5*this%hunger
  end subroutine eat
  subroutine play(this)
    type(Animal_type) :: this
    this%happiness=50+0.5*this%happiness
  end subroutine play
  subroutine makeNoise(this)
    type(Animal_type) :: this
    print *,'woofoink!'
  end subroutine makeNoise
end module Animal_module
```

Encapsulation

To make and manipulate Animal objects:

```
use Animal_module
type(Animal_type) :: lassie, babe
call eat(lassie)
call play(babe)
call makeNoise(lassie)
```

Encapsulation

Note the differences between the c++ and the Fortran versions:

- Fortran is much more verbose (19 lines of code versus 10)
- c++ provides a more natural and concise syntax.
- In c++, the functions eat, play, etc. are considered as part of the object.
- In Fortran, eat, play, etc. are just globally-available functions that take an argument of type `Animal_type`.

Encapsulation

A more concrete example of this type of encapsulation is the `distributed_vector` type that was introduced into the IFS in CY18R2. (Eventually superseded by the `control_vector` type).

`Distributed_vectors` provided a 1-dimensional vector that hid their internal structure (in particular, the fact that the elements are distributed over processors), and provided a clean interface to the rest of the code:

A particular feature of `distributed_vectors` was their use of operator overloading. This allows us to write

```
x = d1 + d2
```

instead of

```
call add_distributed_vectors(x,d1,d2)
```

Encapsulation

`Distributed_vector` Example:

```
use distributed_vectors
type(distributed_vector) :: d1, d2
real :: x(10000), y(10000)

call allocate_vector (d1,10000)
call allocate_vector (d2,10000)

d1 = x  ! Distribute the elements of x among processors.

d2 = 5.0*d1 - 3.0*d2          ! The calculations here are
d1 = d2/dot_product(d1,d2)    ! parallelised, and the dot
                              ! product involves message
                              ! passing.

y = d1  ! This statement does message passing to gather
        ! the distributed elements onto one processor.
```

Encapsulation: Summary

- Although a full object-oriented language provides a more natural framework, simple objects and encapsulation are perfectly possible in Fortran, through proper use of modules.
- Several examples have made their way into the IFS in recent years through the increasing use of derived types:
  - GFL structures, Control_vectors, Distributed_vectors, Spectral_fields, etc.
- However, most modules in the IFS are glorified `common` blocks, with globally visible contents and no attempt at encapsulation.

Outline

Inheritance

Inheritance allows more specific classes to be derived from more general ones. It allows sharing of code that is common to the derived classes.

In the Animal example, we might want to have a separate class for Dog, to express the fact that dogs are intelligent, and can rescue small boys from disused mine shafts.

However, we also want dogs to use the Animal code for eating and playing.

In c++, we can do this:

```
class Dog: public Animal {
private:
  double intelligence=50.0;
public:
  void rescueTimmy() { ... };
};
```

ECMWF

Inheritance

We can then do this:

```
Dog lassie;
Animal babe;
lassie.eat();
babe.makeNoise();
lassie.rescueTimmy();
```

- lassie.eat() and babe.makeNoise() use the functions defined in the Animal class.
- Only lassie can be told to rescueTimmy(), because only dogs have this specialised ability.

Inheritance

Fortran 90/95 does not support inheritance, but it can be approximated. We define a derived type that contains a component of type Animal:

```
module Dog_module
  use Animal_module
  type Dog_type
    type(Animal_type) :: the_animal
    real :: intelligence=50.0
  end type Dog_type
contains
  subroutine rescueTimmy (this)
    type(Dog_type) :: this
    ...
  end subroutine rescueTimmy
end module Dog_module
```

(NB: In Fortran 90/95, the contents of a derived type are either all public or all private. I need the_animal to be public, so I had to make intelligence public, too.)

ECMWF

Inheritance

Things get a bit ugly when we want to access the functions from the Animal "class":

```
type(Dog_type) :: lassie
type(Animal_type) :: babe
call eat(lassie%the_animal)
call makeNoise(lassie%the_animal)
call eat(babe)
call makeNoise(babe)
```

Clearly, long strings of inheritance could get a bit ridiculous:

```
call multiply( &
& lassie%the_mammal%the_quadruped%the_animal%the_being)
```

whereas, in c++, we would just do this:

```
lassie.multiply();
```

ECMWF

Inheritance: Summary

- Inheritance is *just about* possible in Fortran, but quickly gets ugly.
- Essentially, we have to fake "`is a`" relationships (Lassie is a Dog) by using "`has a`" relationships (Lassie_type contains an Animal_type component).
- Fortunately, this type of simple taxonomy tree is of limited use in our applications.
- The real power of inheritance comes when it is combined with Polymorphism and Abstraction.

Outline

Polymorphism

Polymorphism refers to the ability to re-use a piece of code with arguments of different types.

For example, suppose cg implements a conjugate-gradient minimisation algorithm for vectors defined as real arrays of dimension 10000:

```
subroutine cg (costFunction,startPoint)
external costFunction
real, dimension(10000) :: startPoint
...
end subroutine cg
```

Mathematically, the algorithm does not change if we we replace the array argument by a `distributed_vector`, a 2-D array, a vector of observations, or a spectral field.

Ideally, we would like to have a single code that can be applied to all these different types of arguments.

To return to our Animal example, suppose we have written many lines of code, including hundreds of calls to the makeNoise() function.

We now want to make our application a bit more realistic by making dogs say "woof!" and pigs say "oink!".

In c++, all we have to do is:

```
class Dog: public Animal {
public:
  void makeNoise() { cout << "woof!" << endl; }
};

class Pig: public Animal {
public:
  void makeNoise() { cout << "oink!" << endl; }
};
```

The rest of the code stays the same.

In fortran, we would like to do something like this:

```
type Dog_type
  type(Animal_type) :: the_animal
  character(len=5) :: my_noise="woof!"
end type Dog_type

type Pig_type
  type(Animal_type) :: the_animal
  character(len=5) :: my_noise="oink!"
end type Pig_type

subroutine makeNoise(this)
type(???????) :: this
print *, this%my_noise
end subroutine makeNoise
```

This doesn't work because this must be given a specific type (Animal or Dog or Pig) at compile time.

It *is* possible to implement polymorphism in Fortran 90/95, by making the base type contain pointers to all possible derived types (see, e.g. Decyk, Norton and Szymanski, Computer Physics Communications, 115(9), 1998):

```
type Animal_type
...
type(Dog_type), pointer :: dog_pointer
type(Pig_type), pointer :: pig_pointer
end type Animal_type

subroutine makeNoise(this)
type(Animal_type) :: this
if (associated(this%dog_pointer)) then
  print *,this%dog_pointer%my_noise
else
  print *,this%pig_pointer%my_noise
endif
end subroutine makeNoise
```

But, it is not pretty! NB: Adding a new child type (e.g. Cat_type) requires changes to the base type and its functions.

ECMWF

Polymorphism: Summary

- Polymorphism is difficult in Fortran, but can partially be achieved through clever use of pointers and interface blocks.
- Adding a new possibility (e.g. a Cat) is straightforward in c++.
- In Fortran, it requires changes to the base type (Animal), and to all the interface functions that take Animal arguments.
- Polymorphism can also be implemented in Fortran using the Forpedo pre-processor.
- Full object-oriented languages provide a much more natural means to implement polymorphic code.
- There are several examples where polymorphism could be useful in the IFS:
  - Minimisation algorithms applied in model space and observation space.
  - Spherical transforms applied to distributed and single-processor fields.
  - I/O routines applied to spectral and gridpoint fields.
  - Observation processing of different types and subtypes.

ECMWF

Outline

ECMWF

Abstraction

Abstraction refers to the ability to write code that is independent of the detailed implementation of the objects it manipulates. It allows algorithms to be coded in a manner that is close to their mathematical formulations.

Take 4D-Var, for example. Mathematically, we have:

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}-\mathbf{x_b})^{\mathrm{T}}\mathbf{B}^{-1}(\mathbf{x}-\mathbf{x_b}) + \frac{1}{2}\sum_{\mathbf{i=0}}^{\mathbf{M}}(\mathbf{y_i}-\mathbf{H_i x_i})^{\mathrm{T}}\mathbf{R_i}^{-1}(\mathbf{y_i}-\mathbf{H_i x_i})$$

We can describe the algorithm in terms of a few objects: vectors and matrices in model and observation space, and operators $\mathbf{H_i}$ (and $\mathcal{M}_i$).

These objects have a few simple properties: vectors can be added and subtracted and their dot-product calculated, $\mathbf{H_i}$ can be applied to a model-space vector and produces an observation-space vector, etc.

Abstraction

We should be able to write our algorithms in an abstract way, using objects for which a minimal set of fundamental properties is assumed.

The description of the algorithm should be entirely independent of the code that implements these properties.

An algorithm written in this way can then be applied to any set of objects that implements the fundamental properties.

Thus, for example, once we have coded a 4D-Var algorithm, we should be able to apply it without modification to a wide range of models.

Only object-oriented languages provide the tools required to do this.

Abstraction

Example: In c++, we might implement a Vector class as follows:

```
class Vector {
public:
  virtual Vector & operator=(Vector &)=0;
  virtual Vector & operator+(Vector &)=0;
  virtual Vector & operator-(Vector &)=0;
  virtual Vector & operator*(double &)=0;
  virtual double & dot_with(Vector &)=0;
};
```

Although the syntax is a little alarming, this simply amounts to a mathematical definition of vectors: specifically, that they can be added, subtracted, multiplied by a scalar and the dot-product calculated.

The "virtual" and "=0" parts of the definition indicate that the precise implementation of these properties will be given later, in a class derived from Vector.

Abstraction

Although we have not specified exactly how to add, subtract, (etc.) Vectors, we can still write (and compile) code to manipulate them:

```
Vector u, v, w;
double d;
u=v;
w=u+v;
w=u-v;
d=u.dot_with(v);
u=d*w;
```

Using this technique, we can for example, code the entire 4D-Var algorithm in terms of a handful of abstract classes (State, Increment, Departure, etc.), each of which has a few abstract properties.

To apply the code to a particular model (Lorenz 3-variable, QG, IFS, etc.), we simply define a set of derived classes that implement these abstract properties. The 4D-Var code remains identical for all the models

Abstraction: Incremental 4D-Var on One Slide!

```
void incremental_4dvar(CostFunction4dvar & J,
                       ControlVariable & x,
                       Observation & y,
                       int & nouter ) {

  ChangeVariableSqrtCovar chavar(1, *J.B);
  double zj0, zj1
  int jout;
  int ctlsize = J.B->cvecsize();
  ControlVector dx(ctlsize), gx(ctlsize),
                da(ctlsize);
  dx = 0.0;
  da = 0.0;
  Trajectory traj(J.hmop4d->get_nstep());

  for (jout=0; jout < nouter; jout ++ ) {

    Departure * ydep;
    ydep=J.get_R()->get_dep("ombg");

    Observation * yeqv;
    yeqv=y.clone("obsv");

    // Setup trajectory and departures

    ControlVariable xwork(1,x.get()[0]);
    J.get_hmop4d().nl(xwork,*yeqv,traj);
    ydep->diff(*yeqv,y);
    if (jout == 0) ydep->putdb();
    traj.set(da);
    traj.set(*ydep);
    J.settraj(traj,chavar);

    // compute inital cost and gradient
    dx = 0.0;
    J.simul(dx,gx,zj0);

    // CG Minimization
    CG(J,dx,gx,4);

    // Compute final cost and gradient
    J.simul(dx,gx,zj1);

    // Form increment and analysis
    // in physical space
    Increment * dxtmp;
    dxtmp=J.get_B()->get_inc();
    IncrementalControlVariable xinc(1,*dxtmp);
    chavar.vect2var(dx,xinc);
    *xinc.get()=*xinc.get()+*x.get();
    da = da+dx;
  }

  // Final diagnostics
  ControlVariable xwork(1,x.get()[0]);

  Observation * yeqv;
  yeqv=y.clone("obsv");
  J.get_hmop4d().nl(xwork,*yeqv,traj);
  Departure * ydep;
  ydep=J.get_R()->get_dep("oman");
  ydep->diff(*yeqv,y);
  ydep->putdb();
}
```

ECMWF

Abstraction

A typical abstract type:

```
class Increment {
public:
    Increment(){};
    virtual Increment * clone()= 0;
    virtual Increment & operator=(Increment &)=0;
    virtual Increment & operator=(const double &)=0;
    virtual Increment & operator*(const double &)=0;
    virtual Increment & operator+(State &)=0;
    virtual double dot_product(Increment &)=0;
    virtual void read(const char *)=0;
    virtual void write(const char *)=0;
    virtual Increment & propagate_tl(ModelTrajectory &)=0;
    virtual Increment & propagate_ad(ModelTrajectory &)=0;
};
```

Abstraction

The code on the last two slides is the same for all models, from Lorenz '63 to IFS.

To plug a given model into the system, we must provide derived classes for each of the abstract classes, which implement the virtual functions defined in the abstract classes.

For example, Increment requires us to specify 11 functions. For the QG model, most of these are trivial (4–10 lines of code), so that the derived type QgIncrement is 100 lines of code.

In QgIncrement, the functions propagate_tl and propagate_ad call Fortran routines to perform a timestep of the tangent-linear and adjoint model, respectively.

Nearly all the computational work is done in the Fortran code.

ECMWF

Outline

ECMWF

Conclusions

- Several techniques of object-oriented (OO) programming can be emulated in Fortran 90/95.
- However, the techniques are not natural to the language, and require some rather convoluted code.
- Changes (e.g. adding a new class) that can be achieved with a few localised modifications in C++ may require widespread changes in Fortran 90/95.
- A significant clean-up of the IFS could be achieved without requiring OO techniques by proper use of modules and encapsulation.
- However, the full power of an OO language is required if we want to separate our algorithms from their implementation and produce a fully flexible, model-independent code.
- A flexible code is vital if we are to evaluate and implement new ideas in data assimilation.

ECMWF