The Transition of Numerical Software: From Nuts-and-Bolts to Abstraction

Barry F. Smith [1]

May 29, 1998

[1]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4844. bsmith@mcs.anl.gov, `http://www.mcs.anl.gov/petsc/petsc.html`. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Abstract

Traditionally, much of the numerical analysis community has focused on issues of error bounds, convergence rates and time and space complexity of numerical algorithms, not on robust, reusable software implementations. Until recently, many of the computer science advances in software engineering rarely filtered into numerical codes.

With the growing complexity of (1) parallel computers, (2) numerical algorithms, and (3) application physics that must be modeled, the community has finally begun to realize the need for new techniques to manage the complexity of the codes. Thus, the entire community has entered a learning curve; some groups are well advanced on the curve and develop and use sophisticated class libraries and frameworks, while others are still struggling with dynamic memory allocation and data structures.

This article will discuss some of the issues involved in the transition from a Fortran 77 nuts-and-bolts approach to developing numerical code to a higher level, abstract, object-oriented methodology and how our PETSc development team is trying to ease that transition. In addition, it will discuss two limitations in the Fortran 90 syntax that make it difficult to take full advantage of abstraction when programming purely in Fortran 90.

0.1 Introduction

Numerical algorithms are implemented today using a wide range of programming languages and environments: from Fortran 77 to C, C++, Matlab, Mathematica, and now Java. But much of the heavy lifting is still done using Fortran 77. This is due to a variety of both technical and sociological reasons. This paper will discuss some of the issues involved in the transition from a very concrete, procedural-oriented coding style (as exemplified by most numerical Fortran 77 and C code), to more abstract coding techniques.

As software engineering has evolved, like any discipline, it has developed a large vocabulary of specialized terms. These terms are extremely valuable because they allow efficient, accurate communication of highly involved concepts, but, unfortunately, they form a barrier to large numbers of Fortran programmers. Essentially, one has to learn the vocabulary and understand the concepts well, before being able to efficiently apply them to their problems. Many people feel they cannot afford the "non-productive" months spent learning the new techniques.

The PETSc software package for the scalable solution of partial differential equations being developed at Argonne National Laboratory is intended to allow the gradual transition from traditional Fortran coding to more object-oriented techniques. Our intention is that one need not throw away many years worth of coding and start again from scratch using new, not yet well understood, programming methodologies. Instead, one "morphs" the code over time, in small increments. The final code will often consist of portions that are virtually identical to the original code (usually the computational kernels) and other portions that appear dramatically different (generally the control structure). The goal is for Fortran programmers to, over time, learn the powerful new techniques while continuing to productively generate code, thus eliminating much of the "down-time" associated with jumping over the conceptual barriers in a single step.

PETSc is programmed completely in ANSI C and may be used from Fortran, C, and C++. This article focuses on its support for abstract programming in Fortran 77 and C. Although PETSc support for upgrading (and parallelizing) legacy Fortran 77 code is crucial, our key focus is also on developing a modern environment for writing completely new codes.

In order to limit the size and scope of this article, I have decided to focus on what I consider to be the crucial ingredients in applying some of the modern software engineering techniques to numerical software. Thus I will gloss over much of the details, try to limit the vocabulary I introduce, and skip many points that others may feel are equally, or perhaps, even more important. In particular, I will not discuss *templates* at all. Based on personal experience, it is difficult to appreciate the full power of these ideas until they have been thoroughly absorbed. No single article, book, or even series of books can be a substitute for actually learning the concepts by using them.

A more specifically C++ oriented introduction to these ideas is [BL96]. An interesting set of opinions on C++ object-oriented programming is contained

in [KM96],

0.2 Seven Evolutionary Steps

The learning curve can be (somewhat arbitrarily) divided into seven conceptual stages. No single step is revolutionary; each is a natural progression of the previous; but taken as a whole, they represent a fundamentally different way of thinking about programming numerical methods (in fact, programming in general). The concepts associated with the steps are

1) pointers,

2) dynamic memory management,

3) user defined data-structures,

4) data encapsulation, opaque objects,

5) polymorphism,

6) inheritance and composition, and finally

7) a paradigm shift away from procedural programming.

In the following seven sections I will discuss these steps in general and then mention how the PETSc package provides appropriate support for Fortran 77 and C code.

0.2.1 Pointers

A pointer is simply a variable that contains, not a value, but rather a memory address at which a value (or array of values) is stored. Pointers are familiar (in a hidden form) to all Fortran programmers, for example, when one does

```
double precision a(10,10)
...
call afunction(10,10,a)
...

-----------
subroutine afunction(m,n,a)
integer m,n
double precision a(m,n)
...
```

Fortran does not copy all the values in the array into the subroutine. It merely passes into the subroutine the address at which the values in the array are stored This is an example of passing by pointer (also called passing by reference).

The limitation in Fortran 77, which is somewhat overcome in Fortran 90, is that there is no concept of a pointer to a pointer. This is unfortunate because it means that there is, for example, no way to pass out of a subroutine an array that was created in the subroutine. For example, in C or C++ one could write:

```
double *a;

afunction(&a);
a[5] = 23.0;
....

int afunction(double **a) {
  ...
  /*
    allocate space for an array;
    put into a the address of that space;
  */
  ...
}
```

Note that the symbol * in front of a variable in C and C++ indicates that the variable is a pointer; that is, it contains an address, not a value. The & operator returns the address of any variable (including a pointer variable). Thus, in this case, `&a` is a pointer to a pointer.

In order to access arrays allocated in C from Fortran, there is a standard trick that often works to allow one to pass an "address" back to the calling routine, but it is slightly cumbersome and not portable across all Fortran compilers. The PETSc routines `VecGetArray(), ISGetIndices()`, etc. allow Fortran users to access the arrays generated by PETSc using this technique.

Fortran 90 introduced the explicit concept of pointers to Fortran, but with one very unfortunate limitation: at compile time all routines that involve a pointer must know exactly what type of data the pointer references; that is the compiler has to have access to the module that defines the derived datatype. Below, when I discuss data encapsulation and opaque objects, it will become apparent how extremely limiting this seemingly harmless constraint is. Note that the C compiler will still perform type-checking on the arguments, so the Fortran 90 compiler does not actually provide any stronger error checking in this regard.

Though understanding pointers is crucial to the learning curve, it is interesting to note that at some higher level it is possible to remove the concept of pointers. For example, the JAVA programming language does not have pointers. The JAVA designers eliminated pointers from the language because they can be dangerous, since they allow reading and writing directly to any memory location. JAVA does provide dynamic arrays which are essentially array pointers with built-in bounds checking. This allows accessing array elements without the danger of accessing inappropriate data.

0.2.2 Dynamic Memory Allocation

Once one has a language syntax that supports pointer variables, it becomes possible to consider allocating memory space at run time, rather than only at compile time in pre-dimensioned arrays. In C, this is done in a very simple nuts-and-bolts manner. A call to `malloc()` returns a pointer to a space (of requested size) in memory that may be then used for any purpose. C++ offers the command `new()` which is not so close to the metal; it allocates memory and automatically builds any data structure (technically an array, `struct`, or `class`) that a library has previously defined.

Memory allocation is nice because it allows one to set array dimensions at run-time rather than compile time; but its **real power** is in supporting the next five stages in programming abstractions.

0.2.3 User-Defined Data Structures

In Fortran 77 you often see calling sequences such as

```
subroutine asolvg(m,n,ia,ja,a,x,b,work1,work2,...)

integer m,n,ia(*),ja(*),work1(),...
double precision a(*),x(*),b(*),work2(*),...
```

This is because the language syntax gives no convenient way to gather several related variables into a single item (except in common blocks). Thus, many Fortran 77 routines that are actually conceptually very simple, look complex.

Data structures allow the organization of different types of data together as a single entity. At the simplest level they can be used to shorten calling sequences and thus make code more manageable. For example, in C you may represent a sparse matrix (in compressed sparse row format) as

```
typedef struct {
  int    m,n;
  int    *ia;
  int    *ja;
  double *a;
} SpMat;
```

and call a function whose calling sequence is

```
solve(SpMat *mat,double *x, double *y, ...);
```

Fortran 90 also provides the capability to construct new data structures, called *derived types*. Unfortunately, it has one seemingly minor constraint, that we'll see below has a very large impact: Fortran data structures cannot contain pointers to functions.

Again, like pointers and dynamic memory allocation, the ability to construct new problem-dependent data structures, though very powerful, does not fundamentally change the way one views computer programs. What has been

discussed so far is **not** object-oriented programming, nor is it revolutionary. But building on these three techniques has led to a software revolution; the concepts of object-oriented programming.

Essentially, Fortran 90 and most traditional uses of C take advantage of pointers, dynamic memory allocation and user defined data-structures, but do not go beyond this.

0.2.4 Data Encapsulation and Opaque Objects

Once one is able to create data structures, such as sparse matrices, one can then ask: Is it possible to write most of my code so that it works independently of the specifics of the data structure used to store the sparse matrix? This concealing of data structures from portions of the code is called *data encapsulation* (also *data hiding*).

To make this concrete consider a specific example. Say you wish to solve a linear system using a non-preconditioned Krylov subspace method; thus, in the linear system solve, **only** the matrix-vector product routine need be aware of how the sparse matrix is stored. (Here I do not discuss how to get the matrix values into the encapsulated data structure.) To accomplish this in C one could define in a **public** include file (a file included by all functions)

```
typedef struct _SpMat *SpMat;
```

This tells the compiler that you have (somewhere else) defined a data structure called `_SpMat` that variables of type `SpMat` point to. It gives no indication of the contents or organization of the data structure. Meanwhile in a `private` include file define the actual contents

```
struct _SpMat {
  int    m,n;
  int    *ia,*ja;
  double *a;
};
```

Now, only the routines that include the private file are aware of the internal data structure that the sparse matrix is stored in. The other routines just know there is such a structure defined somewhere. So, if one changes the definition of the data-structure, only those routines that include the private file need be changed; or even recompiled.

The variable type `SpMat` is often referred to as an opaque object, because most of the code cannot see inside it.

The syntax of C specifically allows one to make opaque objects; unfortunately Fortran 90, though it does allow pointers, requires that the compiler know exactly what the pointer is pointing to in each routine it is used. Thus one cannot easily construct true opaque objects in Fortran 90. As mentioned above, both languages support type checking that prevent users from passing in the wrong argument type; so the use of opaque objects does allow for argument type checking.

0.2.5 Polymorphism

Data encapsulation allows one to organize a code so that each data structure is visible only to those routines that need direct access to it. This simplifies changing the data structures since only a limited number of routines need to be changed or even recompiled. However, one still needs to link in different code to use a different data structure.

Say I have two different sparse matrix representations `SpMat1` and `SpMat2` and hence two different sparse matrix-vector product routines `Mult1()` and `Mult2()`. I could now write a solver that supports either of these two data structures with calls of the form

```
if (mattype == 1) {
  MatMult1(mat,x,y);
else if (mattype == 2) {
  MatMult2(mat,x,y);
}
/*
   mat is a pointer to the matrix data structure;
   in one of two formats and x and y are pointers
   to arrays
*/
```

whenever I need to perform a matrix-vector product. This is clearly cumbersome. Polymorphism is the concept that the matrix-vector multiplication routine itself automatically handles calling the correct function associated with that particular matrix data structure. Thus, naively, one could write

```
MatMult(void *matin,double *x,double *y)
{
  if (mattype == 1) {
    MatMult1((SpMat1)mat,x,y);
  } else if (mattype == 2) {
    MatMult2((SpMat2)mat,x,y);
  }
  /*
     The (SpMat1) and (SpMat2) above simply indicate
     to the compiler that matrix mat is known to point
     to a data structure in the format of SpMat1 or SpMat2
     respectively
  */
}
```

Now the application code is much cleaner since it always calls `MatMult()`; but this is not extensible. If wanted to add a third matrix data structure, I would have to edit the `MatMult()` routine to add a third conditional.

To support a more extensible polymorphism (the ability to add new data structures and corresponding code without editing or even recompiling the current code) one could have the data structure itself carry the functions that are

required, in this case, the matrix-vector product. This can be implemented in C by having the matrix data structure consist of two parts; a common part and a private (data structure specific) part. In the simplest case the common part might consist only of a pointer to the function that performs the matrix-vector product. For example,

```
struct _SpMat {
  /*
     The variable mult in the data structure below is a
     pointer to a function that takes an SpMat and two double
     precision arrays as input. The library can call this
     function directly to evaluate the matrix-vector product
  */
  int (*mult)(SpMat,double*,double*);
  void *private;
};
```

Recall

```
typedef struct _SpMat *SpMat;
```

defines `SpMat` to be a pointer to the `_SpMat` data structure. The "universal" matrix multiply routine can be written as

```
int MatMult(SpMat mat,double *x,double *y)
{
  /*
      This calls the function pointer with arguments
      mat, x, and y
  */
  return (*mat->mult)(mat,x,y);
}
```

and the two specific multiply routines might look like

```
/* compress sparse row format */
int MatMult1(SpMat mat,double *x,double *y)
{
  /*
      Here we know that mat->data points to a
      data-structure of type SpMat_1 since this
      function, MatMult1(), was called.
  */
  SpMat_1 *mat_1 = (SpMat_1 *) mat->data;
  int     n = mat_1->n, m = mat_1->m, *ia = mat_1->ia;
  int     *ja = mat_1->ja;
  double  *a = mat_1->a;
```

```
  for ( i=0; i<m; i++ ) {
    y[i] = 0.0;
    for ( j=ia[i]; j<ia[i+1]; j++ ) {
      y[i] += a[j]*x[ja[j]];
    }
  }
  return 0;
}

/* compress sparse column format */
int MatMult2(SpMat mat,double *x,double *y)
{
  SpMat_2 *mat_2 = (SpMat_2 *) mat->data;
  int     n = mat_2->n, m = mat_2->m, *ia = mat_2->ia;
  int     *ja = mat_2->ja;
  double  *a = mat_2->a;

  for ( i=0; i<m; i++ ) y[i] = 0.0;
  for ( i=0; i<n; i++ ) {
    for ( j=ia[i]; j<ia[i+1]; j++ ) {
      y[ja[j]] += a[j]*x[i];
    }
  }
  return 0;
}
```

The beauty of this approach is that it is completely extensible. To add a new matrix data structure **does not require changing a single line of previous code, or even having access to the other source code or data structures.** In addition, data encapsulation and polymorphism eliminate the need for *reverse communication.*

PETSc handles data encapsulation and polymorphism in exactly the manner outlined above. Direct support for Fortran 77 is provided by representing the opaque objects as integers in Fortran and converting them to C pointers in a Fortran interface stub. For example, the matrix-vector multiplication routine called from Fortran could be written as

```
void matmult_(int *mat,int *x,int *y, int *__ierr ){
  *__ierr = MatMult( (Mat) PetscToPointer( *mat ),
                     (Vec) PetscToPointer( *x ),
                     (Vec) PetscToPointer( *y ));
}
```

This converts each of the Fortran integers to the appropriate C pointer and then calls the C routine. Note that, unlike in the previous example code, both vectors and matrices in PETSc are treated as abstract objects; i.e. a vector is not simply an array of values.

C++ includes in the language syntax a more direct way of implementing polymorphism through the use of *virtual functions* in what are called *classes*, which are essentially data structures that contain both code and function pointers. So the syntax to do this in C++ versus C is different, but the fundamental concepts are identical. C++ classes cannot easily be used directly from Fortran or C.

Polymorphism is sometimes called *function overloading*. A related concept is *operator overloading* which allows you to not only have a single function (such as `MatMult()`) call different underlying routines depending on the data structures, but also the symbols `+, -, *, /` etc. C++ provides support for operator overloading, C and JAVA do not. Even Fortran 77 has some limited built-in function and operator overloading; when you call the function `abs()` it calls the correct function depending whether its argument is an integer or single precision, double precision, or complex number. Similarly one uses `a + b` regardless of whether `a` and `b` are integers or floating point numbers. The compiler ensures that the correct operation is performed.

0.2.6 Inheritance and Composition

Data structures that contain their own functionality (via function pointers in C, or virtual functions in C++) are often referred to as *objects*. So defining the interface (calling sequences of the virtual functions) and then implementing one or more data structures and code to provide the functionality can be viewed as building objects: data structures plus code to act on those data structures.

One could, of course, always build new objects from scratch, defining the complete set of virtual functions, etc. But, as when building anything, it is often convenient to build off already existing objects. This can be done in a variety of ways. The simplest is to create a new object by including in it a previously defined object. For example, I could define a grid vector object to be a new object that contains both a grid and a vector. This is an example of `composition`.

`Inheritance` is when one creates a new object by adding properties or data to an existing object. For example, from the general class of sparse matrices, one could add the property of symmetry and have a symmetric sparse matrix object.

The syntax in C++ provides a natural way to create new objects via inheritance and composition. C does not provide any syntax to help in this process. In PETSc we provide support for composition by allowing any object to contain pointers to other objects using a mechanism very similar to the attributes in MPI. This is currently done via calls to `PetscObjectCompose()`.

PETSc has an object called a grid-vector, `GVec`, that is obtained by composing a grid object with a vector object. Grid-vectors can be treated as regular vectors, for exampling performing arithmetic on them or multiplying them by a matrix. In addition, one can perform grid specific operations such as visualization and evaluating grid-based functions.

0.2.7 Paradigm Shift Away from Procedural Programming

Traditional numerical computing (even on parallel machines) has been almost exclusively thought of in terms of a single conceptual stream of computations. One provides a main program that calls subroutines, that may themselves call subroutines, etc. The flow of control of the program is always determined by a single (conceptual) program counter.

With objects, one can think about a very different paradigm of computing. Instead of thinking about control passing between subroutines, one thinks about objects interacting with each other. Objects make requests of other objects and service requests from other objects. For example, in PETSc the nonlinear solver object, denoted by `SNES`, requests a linear solver object, denoted by `SLES`, to perform an approximate linear solution. The linear solver object services the request from the nonlinear solver object. The linear solver object makes requests of the matrix object and vector objects in the process of solving the system. In a highly sophisticated simulation, one could see dozens of these various objects interacting.

The cynic could argue this is only playing with language; after all, underneath this there is still a Von Neumann architecture. This is true, however, I am coming to believe that accepting the paradigm shift is enormously liberating. Intellectually one is no longer strait-jacketed to floating down the river performing calculations along the way. One can think about side calculations cleanly without muddying the water. I give one example below.

Consider a large-scale, three-dimensional fluids calculation that you wish to visualize in real time. Before passing the field data to the graphics pipes it will have to be *reduced* to a level that the hardware can render in real time. In a procedural style solution you could, at every timestep in the fluids code, call a reduction routine and from that stream the reduced data to the graphics engine. This is a fine solution. A drawback is that the fluid solver has to be conscious of the visualization; it has to call the reduction routine! In an object-based model, the reduction object would request the current solution data from the solver object. This is a subtle but crucial difference; the solver control structure (which doesn't know or care a hoot about the visualization) now is completely separated from the visualization control structure.

Consider a slightly more complicated situation that perhaps more clearly indicates the potential advantage. Assume person one is monitoring the solution using a PC connected to the simulation with a low bandwidth. He or she sees something of interest and contacts person two who, on the fly, creates a new data reduction object for a high resolution three-dimensional visualization of the data. Person two monitors for a few minutes and then destroy the high resolution data reduction object. Again, a custom application could support this type of interaction but the control structure in the application code would become rather complicated.

Objects allow one to encapsulation not only data, but also control. There is no longer a single river of control for the program, rather a separate stream of control for each object. This simpifies programs that previously would have

required a great deal of centralized control structure. Each object is instead responsible for its own, more or less, straightforward control.

0.3 Concluding Remarks

In this article I've tried to summarize seven major steps in the learning curve from classical, procedural-oriented numerical programming to the more abstract, objected-oriented techniques. In addition, I've indicated how the PETSc software package provides at least some of the support needed to take advantage of these techniques from legacy Fortran 77 code. I've also tried to indicate why it is difficult to use all seven of these techniques when programming purely in Fortran 90. Specific problems with using Fortran 90 to implement data encapsulation are that all routines that access a Fortran 90 pointer have to know the type of data structure it is pointing to and implementing polymorphism is difficult because Fortran 90 data structures cannot contain function pointers.

The complete PETSc distribution is freely available at `http://www.mcs.anl.gov/petsc/`. PETSc is part of the Advanced Large-scale Integrated Computational Environment (ALICE) [MCSD97], being developed in the Mathematics and Computer Science Division at Argonne National Laboratory.

Other object-oriented numerically focused software systems include:

- Blitz++, http://monet.uwaterloo.ca/blitz/,
- Diffpack, http://www.nobjects.com/prodserv/diffpack/,
- ISIS++, http://www.ca.sandia.gov/isis/,
- POOMA, http://www.acl.lanl.gov/PoomaFramework/,
- Overture, http://www.c3.lanl.gov/ henshaw/Overture/Overture.html,
- OPlus, http://www.comlab.ox.ac.uk/oucl/oxpara/parallel/oplus.htm,
- PLapack, http://www.cs.utexas.edu/users/plapack/interface/interface.html.

These packages have varying support for mixing with legacy Fortran code.

0.4 Acknowledgments

I thank my fellow PETSc team members: Satish Balay, Bill Gropp and Lois Curfman McInnes.

Bibliography

[BL96] A. M. Bruaset and H. P. Langtangen. *A Comprehensive set of Tools for Solving Partial Differential Equations: Diffpack.* Birkhauser, 1996.

[KM96] Andrew Koenig and Barbara Moo. *Ruminations on C++.* Addison-Wesley, 1996.

[MCSD97] The Mathematics and Argonne National Laboratory Computer Science Division. ALICE home page. http://www.mcs.anl.gov/alice, December 1997.