INTRODUCTION

Every few years brings a new computer language. Yet most scientific and engineering computing continues to be done in Fortran, a language that has hardly changed since 1977. Here we explore the hypothesis that a new language, C++, is a worthwhile alternative to Fortran.

C++ is based on the C language. The C language has made remarkable strides in the last several years. Formerly, Fortran was the language that was available on most machines and it had the fewest variations from one machine to another. Today, C replaces Fortran in both universality and uniformity. Now more young engineers and scientists learn C than any other language. C overcomes the memory-allocation limitation of Fortran, and offers new features such as data structures and scope. C however was not designed for numerical analysis and it does not handle matrices as conveniently as does Fortran. There has been little mass movement of scientific code from Fortran to C. We (and many others) have overcome some of Fortran's most glaring deficits by calling C-language subroutines and by using our preprocessors for memory allocation and parameter handling. In the past few years, C has become standardized, and new developments are now centered on the C++ language. C++ shares the advantages of C, but it adds features that permit the addition of new, more complex, data types to the language and allows the author to define all operations possible on the new data types. For instance C++ can handle the concept of arrays *at least* as well as Fortran (whereas C cannot.)

We do not predict that Fortran 90 will die and be replaced by C++, but there are some reasons why this might happen: the growth of C++ has been much more rapid. It already has many more users than Fortran 90. We find C++ compilers on all manner of computers (also the free GNU compiler) whereas our workstation manufacturers still do not offer Fortran 90. New parallel architectures present an opportunity for a malleable language like C++ and a risk for static languages.

C++ was especially designed to enable concepts to be isolated, leading to software that is better debugged and more reusable. A particular problem in geophysical inversion problems (see "Processing versus inversion" (PVI), for example, (Claerbout, 1992)) has been that the Fortran language leads to programs where the physical science is closely interwoven with the numerical analysis in the conjugate-gradient solver. One person (the programmer) must handle all the ingredients of both physics and numerical analysis specializations. Such code is challenging to write and even more challenging to debug! Further, in complicated cases, such as with nonlinear operations or linear operators of very high order, we cannot really be sure the code is debugged, which is disheartening.

Our ultimate goals are:

1. Write real code that is clean enough that we can include it in publications where we now use mathematics and pseudocode.

2. Reduce the "overlap expertise" required to merge scientific application code with numerical analysis code.

3. Write linear operators in such a way that they can be used by people other than the original author, and operators written by different people can easily be combined to produce new operators.

As a first step to meeting these goals, Nichols, Dunbar, and Claerbout (1992) took a body of tutorial Fortran-Ratfor code in the geophysical textbook, "Earth soundings analysis: processing versus inversion" (PVI) by Jon Claerbout (1992) and began converting it to C++. This summer, more users/programmers were recruited: The base classes were improved, and many new operators were added. Some examples from Claerbout's book "Basic Earth Interior" (BEI) were converted to C++ form.

The C++ language

To ameliorate the problem of many of the readers of this document knowing little of C and C++, we recommend following learning pathway: Begin from any of the many dozens of C tutorial books concentrating on learning particularly well the ideas of pointer, structure, and typedef. Then switch to the book (Sessions, 1992) which concerns itself mainly with the "class" and "inheritance" concepts, avoiding the full minutia of the C++ language.

Going beyond Fortran 77, the first concept needed is that of a structure. For example, in Fortran, if a single symbol could represent the idea of `(n,(x(i), i=1,n)` then that symbol would denote something like a structure. Another structure might be `(n1,n2, ((x(i1,i2), i1=1,n1), i2=1,n2)`. A numerical analyst thinking about solving the multivariate regression $\mathbf{y} \approx \mathbf{A}\mathbf{x}$ might need to allocate several instances of the structure representing $\mathbf{x}$. Clearly, the solver program written by the analyst should work for all possible structures. In Fortran, the application programmer ends out allocating the temporary arrays needed by the numerical analyst, leading to long, error prone calling sequences. In Fortran the work of the applications programmer and the numerical analyst are typically brought together by subroutines. C also uses subroutine libraries, and additionally it makes heavy use of "include files" that define structures thus serving to better isolate science from numerical analysis. C++ extends the structure concept to a "class," which is a structure containing subroutine pointers. C++ has elaborate provisions for sharing and hiding data, for defaulting and overriding functions. It also uses include files to implement a new method of merging the work of the applications programmer to the numerical analyst called "inheritance" or "derivation" whose explanation is beyond the scope of this brief summary but which should become clearer as you progress through this document.

WHAT IS A SPACE?

The well known word "space" can lead to considerable confusion because there are many entities that are described as spaces, these include continuous spaces, discrete sampled spaces, abstract infinite vector spaces, etc. In our C++ code the "$<$ `type` $>$ `space`" class describes objects that are a discrete sampling of a physical property in some geometric region, e.g. pressure measured on a plane. The word "$<$ `type` $>$" defines the numerical type of the elements of the space, currently `floatspace` is the only supported type, we hope to implement a `complexspace` class soon.

An abstract space in a numerical problem is a collection of values that could be a single physical space (a single instance of the $<$ `type` $>$ `space` class), or an assemblage of physical spaces. In our C++ code such assemblages are called $<$ `type` $>$ `spacearray`, i.e. an array of $<$ `type` $>$ `space` objects. In this paper we will use the word space to refer to the abstract concept and $<$ `type` $>$ `space` or $<$ `type` $>$ `spacearray` when we wish to refer to a particular representation of a space. The $<$ `type` $>$ `spacearray` can be thought of as a generalization of the single $<$ `type` $>$ `space`; in our code a single $<$ `type` $>$ `space` may be automatically converted to a $<$ `type` $>$ `spacearray` containing a single element.

The floatspace class

An object of class `floatspace`, the class to represent a regulary sampled physical space, has two parts. The first part is a multi-dimensional array of numbers representing the sampled data or model. This array contains floating point values. The second part is a set of axes that describe the physical dimensions of region that the samples are taken from.

Every dimension of a `floatspace` has an axis associated with it, which defines an origin, length, and sampling interval (delta) for the axis, as well as a label. Thus, a `floatspace` is meant as a collection of real physical data, rather than an abstract collection of numbers. This is ideal for many scientific applications, as in geophysics.

The publicly accessible member functions that manipulate objects of the class `floatspace` are relatively few, as operators are the main method of transforming a space. The most important member functions are:

- One or more constructors, creates an instance of the class.
- A destructor, destroys an instance of the class.
- norm(), returns the L_2 norm of the space.
- norm2(), returns the square of the L_2 norm.

- dot(), takes the inner product with another (conforming) space.
- Arithmetic operators that add, multiply, etc. by a scalar.
- Arithmetic operators that add, multiply, etc. conforming spaces.
- An indexing operator that accesses the elements of the space.

There are two ways to create a space:

1. The first is to read it from a SEP cube dataset on disk. The SEP cube consists of two parts, an ASCII header file that contains the parameters describing the dataset, and a data file containing the actual data. The parameters are used to initialize the axes of a `floatspace` and the datafile is read to initialize the matrix. A `floatspace` can be constructed by passing it a pointer to an object of class `SepInput`. The SepInput class is a C++ class that encapsulates interaction with a SEP cube dataset (Dulac and Nichols, 1989).
2. The second way to create a space is to create a matrix, using the matrix constructors, and also an Axislist, which is the class that defines the axes for a space. In our applications this method is seldom used, except in generating test cases. The dot-product test routine described in the appendix uses this type of constructor to build spaces containing random numbers.

All the basic arithmetic operations (addition, subtraction, multiplication, and division) are defined for spaces, both by single values and element-by-element by a conformable space. A conformable space is one that has matching shape and axes. If you have one space that is a 10×200 function of space and time it cannot be added to a 10×200 function of velocity and zero-offset-time. Even if two spaces are both 10×200 functions of space and time, they can only be added if the axis starting values and increments match. This high level of error checking is invisible to the user of the class but it helps prevent many errors. Both coding errors and conceptual errors have been detected by these checks.

Assuming space1, space2 and space3 are conformable spaces, the following code fragment demonstrates the manipulation of spaces.

```
space1 = space2 * 2.0 + .001;
space2 = space1 + space3;
```

Note that this code fragment would also be legitimate code for objects that were space-arrays as long as the arrays, and the individual elements were conformable.

The details of the internal structure of a space are hidden from the user and the mathematical operators are provided by the space class. This results in very clean code that is very close to the mathematical representation of the operations required.

Anything else you wish to do to a space is done by an operator. Operators are written in such a way that they have direct access to the private members of a space (the mechanism for doing this in C++ is to make them a "`friend`" class to the space class.) They can directly access the matrix class that is at the heart of a space. This gives operators extra power in manipulating spaces, since operators are designed to be the primary method for transforming spaces.

OPERATORS

A linear operator on a multidimensional sampled space can be defined as an extended matrix multiply. That is, it is a tensor product from one set of dimensions to another set of dimensions. It can sum along any number of dimensions, eliminating those dimensions. Also, it can expand any number of dimensions, adding new dimensions. Matrix multiplication is a particular case of a linear operator which sums along one dimension, and expands along another.

In most cases we do not handle operators as matrices because their application as a matrix multiply would be unnecessarily inefficient (e.g. discrete Fourier transform) or their storage as a matrix would be unnecessarily large (e.g. partial differential equations). Our operators are not represented as matrices — we define them *procedurally* by a computer subroutine pair. The first subroutine is equivalent to the matrix multiplication,

$$\mathbf{A}\mathbf{x}.$$

The adjoint subroutine computes

$$\mathbf{A}'\mathbf{y},$$

where $\mathbf{A}'$ is the (complex conjugate) transpose matrix. $\mathbf{x}$ is a vector in the model space (range) of the operator and $\mathbf{y}$ is a vector in the data space (domain) of the operator.

When we solve a problem involving the operator both the operator, and its adjoint are often required. Solution methods generally fail if the practitioner provides an inconsistent pair of subroutines. This can be checked by using the dot-product test as described in a later section. The coding of any operator is not finished until a dot-product test has been passed.

In our C++ code the base operator class "`foperator`" defines an operator that maps an input `floatspacearray` to an output `floatspacearray`. The derived class "`fop`" defines an additional mapping that maps a single `floatspace` to a single `floatspace`.

An operator only has three public functions

- The constructor
- The Forward() function that is applied to a model space and returns a data space.
- The Adjoint() function that is applied to a data space to produce a model space.

Our operators are procedural operators. When the operator is constructed (initialized), it is told what axes to apply itself to. At this point, it knows what axes, size, label, etc., it is expecting, and what axes it will output. Then, when the operator is applied to a space, it will automatically find the correct axes to apply itself to. Any extra axes will merely be cycled through, applying the same operator to each index in that dimension. This ability to cycle through extra dimensions is built into the class `fopone` described in a later chapter. Any class that is derived from the `fopone` operator class inherits this ability without the author having to write any code to implement it.

Chains and transposes

We often string together two operators to get a third. Because we generally deal with linear operators that are procedurally defined we cannot regard

$$\mathbf{z} = \mathbf{A}\mathbf{B}\mathbf{x}$$

as

$$\mathbf{z} = (\mathbf{A}\mathbf{B})\mathbf{x}$$

If the operators were matricies we could construct a new matrix that was $\mathbf{AB}$. Since they are defined as procedures we must define the composition of the two operators as a sequence of procedures and we must regard it as

$$\mathbf{z} = \mathbf{A}(\mathbf{B}\mathbf{x}).$$

Our Fortran programs interpret this as meaning that first memory must be allocated for $\mathbf{y} = \mathbf{B}\mathbf{x}$; $\mathbf{y}$ must be erased; $\mathbf{y} = \mathbf{y} + \mathbf{B}\mathbf{x}$ should be computed; $\mathbf{z}$ should be erased; and finally $\mathbf{z} = \mathbf{z} + \mathbf{A}\mathbf{y}$ should be computed. Coding all of this is error prone.

In C++, if you wish to form an operator $\mathbf{C}$ that is $\mathbf{A}$ acting on $\mathbf{B}$, you simply write:

```
fopChain C = A * B
```

The class "`fopChain`" encapsulates the behavior of an operator that is a composite of other operators applied in sequence. When the new operator `C` is applied it will act as though

```
C.Forward(x) ≡ A.Forward( B.Forward( x ) )
C.Adjoint(y) ≡ B.Adjoint( A.Adjoint( y ) )
```

The application programmer does not need to do anything to ensure that the adjoint of the composite operator is correctly applied. When a chained operator is applied in the adjoint sense it will be applied with each component operator applied adjoint and applied in the reverse order. This removes another possible source of errors in writing new code.

A similar helper class is available to create an operator that is the adjoint of another operator, it is also very simple to use:

```
fopAdjoint B(A);
```

Application of `B.Forward()` will be equivalent to applying `A.Adjoint()`.

Operator arrays

An operator array is a two dimensional array which contains operators. Applying a `foparray` to a `floatspacearray`:

$$\mathbf{u} = \mathbf{A}\mathbf{v}$$

works like normal matrix multiplication. All three of $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{A}$ are two dimensional arrays (of operators or spaces.)

$$\begin{pmatrix} \mathbf{u}_{11} & \cdots & \mathbf{u}_{1n} \\ \vdots & \ddots & \\ \mathbf{u}_{m1} & \cdots & \mathbf{u}_{mn} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1k} \\ \vdots & \ddots & \\ \mathbf{A}_{m1} & \cdots & \mathbf{A}_{mk} \end{pmatrix} \begin{pmatrix} \mathbf{v}_{11} & \cdots & \mathbf{v}_{1n} \\ \vdots & \ddots & \\ \mathbf{v}_{k1} & \cdots & \mathbf{v}_{kn} \end{pmatrix}$$

$\mathbf{u}$ is a space array that has the number of rows of $\mathbf{A}$, and the number of columns of $\mathbf{v}$. It is not possible to apply an operator array to an array of unsuitable size without generating a run time error. The rows of $\mathbf{A}$ are applied to the columns of $\mathbf{v}$ and the result is summed. As each element of the operator array is applied to the appropriate element of the space array, run time compatibility checks will take place. e.g. $\mathbf{A}_{11}$ and $\mathbf{v}_{11}$ must be compatible. Also the results to be summed must be compatible, the space that results from $\mathbf{A}_{11}\mathbf{v}_{11}$ must be compatible with the space that results from $\mathbf{A}_{12}\mathbf{v}_{21}$.

In C++, this looks like:

```
floatspacearray u = B.Forward(v);
```

This is almost identical to the syntax for applying a single operator to a single space, except here the variables `u` and `v` are of class `floatspacearray` and the operator `B` is of class `foparray`.

A simple example of using arrays is to generate a damped operator from the regular operator. If we wish to calculate the damped solution to the problem

$$\min_x |\mathbf{A}\mathbf{x} - \mathbf{y}|^2$$

We can augment the problem with a diagonal damping matrix and solve:

$$\min_x \left| \begin{pmatrix} \mathbf{A} \\ \epsilon\mathbf{I} \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y} \\ \mathbf{0} \end{pmatrix} \right|^2 \tag{1}$$

The new operator, $\mathbf{A_2}$ can be written as

$$\mathbf{A_2} = \begin{pmatrix} \mathbf{A} \\ \epsilon\mathbf{I} \end{pmatrix}$$

When applied to a space, $\mathbf{x}$ it gives a space-array,

$$\mathbf{A_2 x} = \begin{pmatrix} \mathbf{Ax} \\ \epsilon\mathbf{Ix} \end{pmatrix}$$

When the adjoint is applied to a space-array it gives a space,

$$\mathbf{A}'_2 \begin{pmatrix} \mathbf{y_1} \\ \mathbf{y_2} \end{pmatrix} = (\mathbf{A}', \epsilon^*\mathbf{I}) \begin{pmatrix} \mathbf{y_1} \\ \mathbf{y_2} \end{pmatrix} = \mathbf{A}'\mathbf{y_1} + \epsilon^*\mathbf{I}\mathbf{y_2}$$

All of this is handled automatically by the `foparray` class.

SOLVERS

In our class library we have a class of objects called "`LSSolver`", these are objects that solve the least-squares minimization problem associated with an operator. There are many classes derived from `LSSolver` that use different methods to solve the problem. The solver classes operate at the most abstract level, they take as arguments objects of class `foperator` and `floatspacearray`. Any class derived from these can be used in the solution algorithm.

The author of the operator class and the author who sets up the particular problem do not have to modify any of the solver code, and the author of a solver routine does not need to assume anything about the operator that will be used other than they must be derived from one of the base operator classes.

This design of the operator and space classes separates the jobs of implementing a particular operator and writing a routine that solves the linear problem. The scientists can concentrate on writing code specific to their domain of expertise and then ask the local scientific computing specialist to write a spiffy solver routine for them. In contrast, when writing in Fortran we typically need to write a new solver routine for each new operator that we implement.

Routines such as solvers and dot-product tests can be written once, and then just called with operators and spaces as arguments. One major task that lies ahead is to implement more types of solver routines. We wish to compare different algorithms for non-Hermitian problems, the use of preconditioners and operators for nonlinear optimizations. Since these routines can be written in terms of abstract operator and space classes we will only have to write these routines once, they will then be available for use on any problem. In an ideal world we would hope to collaborate with members of the computer science department in writing these routines. We would be able to provide many examples of real world problems and a framework in which to use them and they would be able to provide new algorithms for solving the problems.

ADJOINTS AND THE DOT-PRODUCT TEST

A great many of the calculations we do in science and engineering are really matrix multiplication in disguise. One goal of this book is to unmask the disguise by showing many examples. Second, we will illuminate the meaning of the **adjoint** operator (matrix transpose) in these many examples. Third we show how to write computer code in C++.

Modeling calculations generally use linear operators that predict data from models. We often need to find the inverse of these calculations, i.e., to find models (or make maps) from the data. Logically, the adjoint is the first step and a part of all subsequent steps in this **inversion** process. Surprisingly, in practice the adjoint sometimes does a better job than the inverse! This is because the adjoint operator tolerates imperfections in the data and does not demand that the data provide full information. Once you grasp the relationship between operators in general and their adjoints, you can have the adjoint just as soon as you have learned how to code the modeling operator.

If you will permit me a poet's license with words, I will offer you the following table of examples of **operator**s and their **adjoint**s:

operator	adjoint
matrix multiply	conjugate-transpose matrix multiply
convolution	crosscorrelation
stretching	squeezing
zero padding	truncation
causal integration	anticausal integration
add functions	do integrals
plane-wave superposition	slant stack
superposing on a curve	summing along a curve
upward continuation	downward continuation
diffraction modeling	imaging by migration
hyperbola modeling	CDP stacking
ray tracing	tomography

The left column above is often called "**modeling**," and the adjoint operators on the right are often used in "data **processing**."

When the adjoint operator is *not* an adequate approximation to the inverse, then you apply the techniques of fitting and optimization which require iterative use of the modeling operator and its adjoint.

The adjoint operator is sometimes called the "**back projection**" operator because information propagated in one direction (earth to data) is projected backward (data to earth model). With complex-valued operators the transpose and complex conjugate go together and in Fourier analysis, taking the complex conjugate of $\exp(i\omega t)$ reverses the sense of time. Still assuming poetic license, I will say that adjoint operators *undo* the time and phase shifts of modeling operators. The inverse operator does this too, but it also divides out the color. For example, with linear interpolation high frequencies are smoothed out, so inverse interpolation must restore them. You can imagine the possibilities for noise amplification. That is why adjoints are safer than inverses. But nature determines in each application what is the best operator to use, whether to stop after the adjoint, to go the whole way to the inverse, or to stop part-way.

We will see that computation of the **adjoint** is a straightforward adjunct to the computation itself, and the computed adjoint should be, and generally can be, exact (within machine precision). If the application's operator is computed in an approximate way, we will see that it is natural and best to compute the adjoint with adjoint approximations.

There is a huge gap between the conception of an idea and putting it into practice. During development, things fail far more often than not. Often, when something fails, many tests are needed to track down the cause of failure. Maybe the cause cannot even be found. More insidiously, failure may be below the threshold of detection and poor performance suffered for years. I find the **dot-product test** to be an extremely valuable checkpoint.

Conceptually, the idea of matrix transposition is simply $a'_{ij} = a_{ji}$. In practice, however, we often encounter matrices far too large to fit in the memory of any computer. Sometimes it is also not obvious how to formulate the process at hand as a matrix multiplication. (Examples are differential equations and fast Fourier transforms.) What we find in practice is that an application and its adjoint amounts to two subroutines. The first subroutine amounts to the matrix multiplication $\mathbf{Ax}$. The adjoint subroutine computes $\mathbf{A'y}$, where $\mathbf{A'}$ is the conjugate-transpose matrix. In a later chapter we will be solving huge sets of simultaneous equations. Then both subroutines are required. We are doomed from the start if the practitioner provides an inconsistent pair of subroutines. The dot-product test is a simple test for verifying that the two subroutines are adjoint to each other.

According to texts on linear algebra the "Hilbert Adjoint" operator is defined as the operator $\mathbf{A'}$ that satisfies the following relationship for any $\mathbf{x}$ and $\mathbf{y}$.

$$< \mathbf{y}\ ,\ \mathbf{A}(\mathbf{x}) > = < \mathbf{A}'(\mathbf{y})\ ,\ \mathbf{x} > . \tag{2}$$

Where $< \mathbf{a}, \mathbf{b} >$ is the inner product of $\mathbf{a}$ and $\mathbf{b}$.

In matrix notation this can be written in the more familiar form

$$\mathbf{y}'(\mathbf{A}\mathbf{x}) \ = \ (\mathbf{A}'\mathbf{y})'\mathbf{x}$$

The vector inner product is the adjoint of the first vector dotted with the second vector.

In our C++ classes the inner product is obtained by using the `dot()` member function of a `floatspace` or `floatspacearray`.

For the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Compute the vector $\tilde{\mathbf{y}} = \mathbf{Ax}$ using your program for $\mathbf{A}$, and compute $\tilde{\mathbf{x}} = \mathbf{A'y}$ using your program for $\mathbf{A'}$. Inserting these into equation 1.2 should give you two equal scalars

$$< \mathbf{y}, (\mathbf{Ax}) > \ = \ < \mathbf{y}, \tilde{\mathbf{y}} > \ = \ < \tilde{\mathbf{x}}, \mathbf{x} > \ = \ < (\mathbf{A'y}), \mathbf{x} > \tag{3}$$

The left and right sides of this equation will be computationally equal only if the program doing $\mathbf{A}'$ is indeed adjoint to the program doing $\mathbf{A}$ (unless the random numbers do something miraculous).

Using the C++ classes we can write a single dot-product test routine that all new operators can be tested with. We only need to write a main program that supplies the operator, and a definition of the data space, as arguments. The test routine will generate spaces of random numbers and check that the dot product test condition is satisfied. The ability to construct chains and arrays of operators allows us to make the following statement; if all the individual operators in the chain or array pass the dot-product test then the composite operator will always pass the dot-product test. Thus a huge family of coding errors possible in Fortran are not possible in C++.

Do not be alarmed if the operator you have defined has **truncation** errors. Such errors in the definition of the original operator should be identically matched by truncation errors in the adjoint operator. If your code passes the **dot-product test**, then you really have coded the adjoint operator. In that case, you can take advantage of the standard methods of mathematics to obtain inverse operators.

We can speak of a continuous function $f(t)$ or a discrete one f_t. For continuous functions we use integration, and for discrete ones we use summation. In formal mathematics the dot-product test *defines* the adjoint operator, so the definition if the inner product is fundamental to the definition of the adjoint operator. The input or the output or both can be given either on a continuum or in a discrete domain. So the dot-product test $< \mathbf{y}, \tilde{\mathbf{y}} >=< \tilde{\mathbf{x}}, \mathbf{x} >$ could have an integration on one side of the equal sign and a summation on the other. Linear-operator theory is rich with concepts, but we will not develop them in this book on its applications and C++ implementation.

In mathematics the word "**adjoint**" has three meanings. One of them, the so-called **Hilbert** adjoint, is the one generally found in physics and engineering and it is the one used in this book. In matrix algebra there is a different matrix, called the **adjugate** matrix. It is a matrix whose elements are signed cofactors (minor determinants). For invertible matrices, this matrix is the determinant times the inverse matrix. It is computable without ever using division, so potentially the adjugate can be useful in applications where an inverse matrix cannot. Unfortunately, the adjugate matrix is sometimes called the adjoint matrix particularly in the older literature. For that reason we often find the usage "conjugate-transpose", "transpose", or "conjugate" in the literature where Hilbert adjoint is what is intended.

REFERENCES

Claerbout, J. F., 1992, Earth Soundings Analysis : Processing Versus Inversion: Blackwell Scientific Publications.

Dulac, J. C., and Nichols, D., 1989, Object oriented programming for seismic data: SEP–**61**, 303–318.

Nichols, D., Dunbar, G., and Claerbout, J. F., 1992, The C++ language in physical science: SEP–**75**, 455–472.

Sessions, R., 1992, Class construction in c and C++: PTR Prentice Hall, New Jersey.