

July 28, 1993

# C++ and Object-Oriented Numerics

Kent G. Budge, James S. Peery, Allen C. Robinson, and Michael K. Wong

Sandia National Laboratories

Albuquerque, New Mexico 87185

## Abstract

C++ is often described as an object-oriented programming language because of its strong support for classes with multiple inheritance and polymorphism. However, for a growing community of numerical programmers, an equally important feature of C++ is its support of operator overloading on value-semantic classes. The union of these two techniques results in a programming style which we choose to call *object-oriented numerics*.

The object-oriented numerics paradigm emphasizes the distinction between *polymorphic classes* and *value-semantic classes*. Polymorphic classes are generally high-level, are part of an inheritance hierarchy, and have an interface of named functions based on message passing. Value-semantic classes are generally low-level, are not part of an inheritance hierarchy (but may form a conversion hierarchy), and have a public interface of overloaded operators based on an algebra. Numerical codes are best structured as a set of polymorphic classes which use value-semantic classes to exchange messages, represent internal states, and perform computations.

Most of the efficiency concerns that have been voiced by numerical programmers about C++ focus on the use of value-semantic classes in computationally intensive code. In fact, C++ translators can generate efficient code for many value-semantic classes. However, for the important case of smart arrays, fundamental difficulties remain. We discuss the two most important of these, namely, aliasing ambiguities and the proliferation of temporaries, and discuss some solutions.

Published in *Journal of C Language Translation*, Vol. 5, No. 32 (1993)

---

This work performed at Sandia National Laboratories supported by the U. S. Department of Energy under contract number DE-AC04-76DP00789.

## Introduction

Booch defines an object-oriented language as one which supports objects having an interface of named operations and a private internal state; which associates a class with each object; and which allows classes to inherit attributes from other classes [2]. The C++ programming language is an example of an object-oriented language which has enjoyed great popularity.

Our experience in writing numerical programs in C++ has shown us the importance of making a distinction between *polymorphic classes*, and *value-semantic classes*. Polymorphic classes are part of an inheritance hierarchy and have a public interface consisting largely of named member functions. Value-semantic classes are not part of an inheritance hierarchy and have a public interface based largely on overloaded operators. C++ is the only popular programming language whose class construct fully supports both polymorphic classes and value-semantic classes.

In this paper, we discuss how both polymorphic classes and value-semantic classes have been used in large numerical programs and the reasons why we believe both are required. We outline the differences between these two broad categories of classes and provide guidelines for determining their correct use. We conclude with a discussion of the pragmatics of compiling numerical programs written in C++.

## Polymorphic Classes

A polymorphic class is part of a public inheritance hierarchy, and its instances are typically manipulated by reference. Because an instance of a polymorphic class can be manipulated through a reference to one of its base classes, it can masquerade as an instance of one of its base classes. In other words, its *apparent* type need not be the same as its *exact* type. We reserve the term *object* to refer to instances of polymorphic classes.

This property of polymorphic classes implies that inheritance hierarchies should model a set of “Is A” relationships. Given a derived class D and its base class B, we can state that D “Is A” B. The use of reference to a D as if it were a B implies that both the internal states and external properties of B form a subset of the states and properties of D. In practice, this is too restrictive; it is common for the internal states of D to differ significantly from those of B even when the external properties are compatible. The solution is to qualify selected member functions of B as *virtual*. These functions in B are dynamically overridden by functions in D that have identical signatures, but differ in their interpretation of the object’s internal state [13].

Another way of viewing virtual functions is as a means to safely cast the type of an object reference for the duration of a single procedure call. The more exact type to which the reference is cast is determined by the object referred to and need not be declared in the calling procedure. Because the newly cast reference type is more exact, the virtual function has access to more of the object’s internal state.

We have found that polymorphic class hierarchies represent certain concepts in numerical codes very well. For example, two of our engineering codes, PCTH and RHALE [9], make use of a material response library (MATRESLIB) which is implemented as a set of polymorphic class libraries [15]. In the equation of state library, a particular equation of state model is implemented as a class derived from the abstract base class `Equation_of_State`. For example,

```
class Equation_of_State {
private:
    double cv; // heat capacity at constant volume
    /* ... */
public:
    virtual double Pressure(double T, double rho) = 0;
    /* ... */
};

class Ideal_Gas : public Equation_of_State {
private:
    double gamma; // adiabatic ratio
```

```

    /* ... */
public:
    double Pressure(double T, double rho);
    /* ... */
};

class Van_der_Waals_Gas : public Equation_of_State {
private:
    double a, b;          // Van der Waals parameters
    /* ... */
public:
    double Pressure(double T, double rho);
    /* ... */
};

```

In this example, the heat capacity  $c_v$  is interpreted differently in the two derived equations of state, even though there is enough universality in the concept to warrant placing it in the base abstraction. It is a constant for an ideal gas, but is regarded as a reference value for a Van der Waals gas where the heat capacity changes with temperature and pressure. This difference in interpretation is incorporated into virtual functions such as `Pressure`. These functions also provide access to additional internal state information (such as the adiabatic ratio  $\gamma$  for the ideal gas or the Van der Waals parameters  $a$  and  $b$  for the Van der Waals gas) for the duration of the function call.

In simulations, materials are represented by instances of the `Material` class. The `Material` class “Has A” pointer to `Equation_of_State`. At run time, an instance of the desired equation of state is dynamically bound to this pointer. Subsequent references to that equation of state can be programmed without any knowledge of the specific equation of state by referring to the pointer. This approach has increased the maintainability and reusability of our material response library by eliminating nearly all case statements and if-tests based on the exact equation of state used. The same approach has been used in MATRESLIB for many other material models, such as strength models, burn models, heat conductivity models, and fracture models.

The class hierarchy resulting from this approach is quite simple and is illustrated in Figure 1. Each concrete class is derived directly from the common base class.

Another example of a useful polymorphic class hierarchy is a set of boundary specification classes. The RHALE code uses a finite element algorithm to simulate strong shock waves in various materials. The finite element mesh includes specifications of boundaries to which particular boundary conditions might be applied. Some boundary conditions, such as kinematic boundary conditions, apply a constraint directly to a set of nodes (the points which define elements). The boundaries to which these conditions are applied are adequately described by a list of node numbers. On the other hand, pressure boundary conditions are applied to element faces rather than directly to nodes. The boundaries to which these conditions are applied must include a description of the element faces as well as the list of node numbers. Finally, contact boundary conditions require complete knowledge of the connectivity of the different element faces on the boundary. These different boundary specifications are illustrated in Figure 2. They form the simple linear inheritance hierarchy shown in Figure 3.

The usefulness of this hierarchy derives in part from the fact that the more complete boundary descriptions are expensive to compute and store. Hence, one prefers to use the simplest description that is adequate. However, if a more complete description already exists for a given boundary, the inheritance hierarchy allows it to be used exactly as if it were a less complete description. This results in a hierarchy in which all the classes are concrete.

Finally, we consider the mesh class hierarchy being developed for RHALE. This hierarchy is illustrated in Figure 4. This hierarchy is more complex than the previous two, since it includes both multiple branches and multiple levels of inheritance and uses some abstract base classes.

The common characteristic of the examples provided above is that they are all complex systems where specialization is achieved by augmentation. This is exactly the hierarchy model supported by polymorphic classes, and we have found C++ to be well-suited for representing these abstractions in numerical software.

## Value-Semantic Classes

Value-semantic classes (also called *concrete classes* or *abstract data types*) differ from polymorphic classes in that they represent well-defined mathematical values. The set of functions associated with value-semantic classes models an algebra rather than a message interface. Polymorphic classes and value-semantic classes can be easily confused, but we believe the distinction is real and important.

An example of a value-semantic class is the complex number class:

```
class complex {
  private:
    double re, im;

  public:
    complex(void);
    complex(double real, double imag = 0.0);

    friend complex operator+(const complex&, const complex&);
    /* ... etc ... */

    double abs(const complex&);
    /* ... ad nauseam ... */
};
```

This class represents a well-defined set of mathematical values (those complex values within the range and precision of the hardware.) It has a well-defined algebra represented by a set of overloaded functions and operators. It is the prototypical value-semantic class and is actually a built-in type in other languages.

Other examples of value-semantic classes are provided by the Sandia National Laboratories PHYSLIB library [3]:

```
class Vector {
  private:
    double x, y, z;
  public:
    friend Vector operator+(const Vector&, const Vector&);
    /* ... etc ... */
};

class Tensor {
  private:
    double xx, xy, xz;
    double yx, yy, yz;
    double zx, zy, zz;

  public:
    friend Tensor operator*(const Tensor&, const Tensor&);
    friend Vector operator*(const Tensor&, const Vector&);
    /* ... etc ... */
};

class SymTensor { // a symmetric tensor
  private:
    double xx, xy, xz;
    double      yy, yz;
    double      zz;

    /* ... etc ... */
};

/* ... etc ... */
```

These classes simplify the coding of tensor calculations in physics codes by providing a convenient representation of tensor algebra. They exist in 1-D, 2-D, and 3-D versions which are differentiated through preprocessor directives (and therefore selected at compile time.) Almost all operations are implemented as inline friend functions, and, with high-quality compilers, it is possible to obtain a numerically efficient executable program.

Like polymorphic classes, value-semantic classes can be arranged into “Is A” hierarchies of increasing specialization. For example, a real number “Is A” complex number with zero imaginary part. A symmetric tensor “Is A” full tensor whose lower triangular components are equal to the corresponding upper triangular components. However, this kind of specialization is *not* well represented by an inheritance hierarchy, since the direction in which the representation is augmented (by adding more components) is *opposite* to the direction in which specialization takes place. This violates a basic assumption of polymorphic class hierarchies, which is that the direction in which representations are augmented is the *same* as the direction in which specialization takes place.

One benefit of inheritance for polymorphic classes is that it lets an object of a derived class D masquerade as an object of a base class B, reducing the number of distinct functions that must be written. These semantics are provided for value-semantic classes through the mechanism of *user-defined conversions*. These conversions permit a value of a specialized type (such as `SymTensor`) to masquerade as a value of a more general type (such as `Tensor`). For example,

```

class SymTensor {                                // a symmetric tensor; specialized
private:
    double xx, xy, xz;
    double yy, yz;
    double zz;

    /* ... etc ... */
};

class Tensor {                                    // a full tensor; general
private:
    double xx, xy, xz;
    double yx, yy, yz;
    double zx, zy, zz;

public:
    Tensor(const SymTensor& a) :
        xx(a.xx), xy(a.xy), xz(a.xz),
        yx(a.yx), yy(a.yy), yz(a.yz),
        zx(a.xz), zy(a.yz), zz(a.zz) {}

    friend double Tr(const Tensor& a){ return a.xx + a.yy + a.zz; }
    /* ... etc ... */
};

main(){
    SymTensor a;
    /* ... */
    double t = Tr(a);
}

```

In this code fragment, `a` is a `SymTensor` that is converted to a `Tensor` to serve as the argument for `Tr(const Tensor&)`. This masquerade mechanism is less efficient than inheritance, since a temporary must be constructed, but its inefficiency is compensated by the savings provided by a smaller representation of the specialized type. In principle, a compiler can optimize many such conversions and function calls, if inlined, by treating a conversion as a set of aliases.

Value-semantic classes can be placed into a *conversion hierarchy* that is analogous to the inheritance hierarchies characteristic of polymorphic classes. Such hierarchies may include builtin types (such as `int`, `double`, or pointers)

since conversion to these types is possible. We illustrate such a hierarchy in Figure 5. A conversion relationship can be distinguished from an inheritance relationship in class diagrams through the use of an unfilled arrow.

Our experience shows that polymorphic classes are rarely useful for value representation, even when the assumption that specialization is achieved through augmentation is not violated. There are two reasons for this. First, values are the building blocks of computation, and their representation and operations *must* be efficient. This generally requires that the exact type of a value be known at compile time. In particular, virtual functions are virtually useless in value-semantic classes. Second, experience shows that *object slicing* is a major problem for classes with overloaded operators that are part of a public inheritance hierarchy [14].

We illustrate object slicing with the following example:

```
class dblarray {
private:
    size_t length;
    size_t *reference_count;
    double *data_array;

public:
    friend dblarray operator+(const dblarray&, const dblarray&);
    /* ... etc ... */
};

class cc_field : public dblarray {           // cell-centered field
private:
    Topology *mesh_topology;

public:
    friend cc_field Laplacian(const cc_field&, const cc_field&);
    /* ... */
};
```

We have specialized a generic smart array class to represent a cell-centered scalar field on a finite-element domain. This is the rare case of specialization by augmentation for a value-semantic class. The additional data member is a pointer to an object which represents the topology of the finite-element domain. This added data supports calculus operations.

Note what takes place, however, if we add two `cc_field` values. The result is a `dblarray`, not a `cc_field`; the topological information has been lost. One can get around this by redefining all the arithmetic operators, but then one loses most of the benefits of inheritance. It is better to move the calculus operations into the `Topology` class and stick with using `dblarray` to represent all array-like values. Polymorphic classes are more easily specialized than value-semantic classes.

An interesting characteristic of value-semantic classes is that it often seems sensible to place the data members in the public interface. In the case of the complex number class shown earlier, the only data members are the `double` values representing the real and imaginary parts. It can be argued that these have an independent meaning useful to programmers and should therefore be accessible. On the other hand, one might decide to change the representation (for example, from real and imaginary parts to magnitude and argument), which will break existing code if the representation is public. This argues for a buffer of access functions between the user and the representation [7]. We choose to make data members public only when the representation is both universally accepted and computationally efficient.

Another characteristic of value-semantic classes is that “Has A” relationships are extremely common. This often results in some uncertainty as to the best way to decompose an aggregate value. Consider three examples:

```
class complex_array {
private:
    size_t length;
    size_t *reference_count;
```

```

    complex *array;

public:
    complex& operator[](size_t n){
        return array[n];
    }
    complex operator[](size_t n) const {
        return array[n];
    }
    /* ... */
};

```

versus

```

class complex_array {
private:
    dblarray real, imaginary;

public:
    complex element(size_t n) const{
        return complex(real[n], imaginary[n]);
    }
    void element(size_t n, const complex& c){
        real[n] = c.real();
        imag[n] = c.imag();
    }
    /* ... */
};

```

versus

```

class complex_array {
private:
    dblarray packed_array;

public:
    complex& operator[](size_t n){
        return *(complex*)&packed_array[n<<1];
    }
    complex operator[](size_t n) const {
        return complex(packed_array[n<<1],
                       packed_array[(n<<1)+1]);
    }
    /* ... */
};

```

The first example implements an array of complex values, whereas the second implements a complex value of arrays. Both classes represent the same concept. This is the “array of objects” versus “object of arrays” dilemma. The jury is still out on which is the best approach.

The third approach is what we might call “object of array.” The individual complex values are stored in a single `dblarray`. This approach is sensible if `dblarray` is highly optimized and an efficient subscript operator is required (ruling out the second approach, where access functions must be used instead.)

In general, it is best to avoid a subscript operator for classes representing arrays of aggregate values. Since internal representations should be flexible, an interface that is insensitive to the representation is preferable. Efficient element access functions can be written for all of the above internal representations.

We summarize the distinctions between polymorphic classes and value-semantic classes in Table 1.

## Why Distinguish Polymorphic Classes from Value-Semantic Classes?

The distinction between these two types of classes is real and important. Each does a particular task best. Polymorphic classes are much too inefficient for low-level, value-based computation. On the other hand, value-semantic classes are not powerful enough to handle complexity at high levels of a program.

We believe that the most effective way to use C++ for numerics is to structure the code as a set of polymorphic classes that use value-semantic classes for message passing, representation of internal states, and computation. This implies that the polymorphic classes exist largely in the upper levels of the code, while value-semantic classes will be found throughout the lower levels.

The computational physicist will regard such a code as an environment in which polymorphic classes provide *context* for representing physics equations as expressions using value-semantic classes. For example,

```
LVectorField Block::Internal_Force(void) const
{
    SymTensorField stress =
        Sym(rotation * material->Stress() * Trans(rotation));
    return linklist->Element_Surface_Integral(stress, curcoor);
}
```

Block, a polymorphic class, represents a portion of the problem domain. It Uses A polymorphic class to represent the material (through the pointer `material`) and another polymorphic class to represent the mesh topology (through the pointer `linklist`). These classes provide all the context necessary to translate an important physics equation into an expression using value-semantic classes such as `SymTensorField` and `LVectorField`.

As we see it, the alternative to programming in this way is to use polymorphic classes in the upper levels of a code, but drop into FORTRAN or FORTRAN-like code in the lower levels. Although one must ultimately define the value-semantic class operations in terms of FORTRAN-like code, the use of value-semantic objects extends the range of applicability of the concepts of data encapsulation and abstraction into much lower levels of a code.

It is not enough that value-semantic classes provide far superior software engineering characteristics to FORTRAN-like code. To be competitive with FORTRAN, the code generated by a C++ translator for value-semantic class expressions must also approach FORTRAN-like efficiency. The remainder of our paper addresses this concern.

## Function Inlining and Value-Semantic Class Operators

Because values are the building blocks of computation, the overloaded operators associated with value-semantic classes *must* translate into efficient machine code. This means that the overload functions should be inlined wherever possible to permit optimizations that transcend the scope of individual operations.

For example, consider the following:

```
class complex {
private:
    double re, im;

public:
    complex(void){};
    complex(double real a, double imag b = 0.0) : re(a), im(b) {}

    friend complex operator+(const complex& a, const complex& b){
        return complex(a.re+b.re, a.im+b.im);
    }
    friend complex operator*(const complex& a, const complex& b){
        return complex(a.re*a.re-a.im*a.im, a.re*b.im+a.im*b.re);
    }
    /* ... etc ... */
}
```

```

};

main(){
    complex a, b, c, x, y;

    /* ... */

    y = a + x*(b + x*c);

    /* ... */
}

```

When the various inline operator and constructor calls are expanded, the resulting code might look something like

```

main(){
    complex a, b, c, x, y;
    /* ... */
    {
        complex tmp1, tmp2, tmp3;
        tmp1.re = x.re*c.re-x.im*c.im;
        tmp1.im = x.re*c.im+x.im*c.re;
        tmp2.re = b.re+tmp1.re;
        tmp2.im = b.im+tmp1.im;
        tmp3.re = x.re*tmp2.re-x.im*tmp2.im;
        tmp3.im = x.re*tmp2.im+x.im*tmp2.re;
        y.re = a.re+tmp3.re;
        y.im = a.im+tmp3.im;
    }
    /* ... */
}

```

which a good optimizer can transform to

```

main(){
    complex a, b, c, x, y;
    /* ... */
    {
        register double re, im, xre=x.re, xim=x.im;
        re = b.re+xre*c.re-xim*c.im;
        im = b.im+xre*c.im+xim*c.re;
        y.re = a.re+xre*re-xim*im;
        y.im = a.im+xre*im+xim*re;
    }
    /* ... */
}

```

The transformed code is more efficient, since it stores intermediate results in registers or caches and avoids redundant memory fetches. Unfortunately, today's translators do not always perform this well. For example, the original implementation of C++ refuses to inline functions that are used more than once in a single expression [13]. However, this is a restriction which could easily be lifted. A translator should certainly be able to inline any function which can be reduced to a comma expression [11]; should not necessarily regard multiple uses of an inline function in an expression as a hint to outline the function; and should be able to inline nested inline function calls. When these conditions are satisfied, most expressions involving value-semantic classes can be translated into efficient machine code.

An interesting feature of overloaded operators on value-semantic classes is that they seem to fall into one of two categories: those which are essentially constructors (such as most arithmetic operators), and those which are access functions (such as the subscript operator). In the previous examples, all the arithmetic operators were wrappers for constructor calls with simple expressions as arguments. Perhaps in some future child of C++ one will be able to write

```

class complex {
    double re, im;

```

```

public:
    constructor(double real, double imag) : re(real), im(imag) {}

    constructor operator+(const complex& a, const complex& b){
        re = a.re + b.re;
        im = a.im + b.im;
    }

    /* ... */
};

```

Written this way, the true nature of the operator is revealed. It constructs a new value from two other values according to the rules of its algebra. The operator is a member function, but preserves the symmetry of its operands. However, since it is not significantly more efficient than present syntax, we do not seriously propose such an extension to C++ at this time; we simply present it as food for thought.

## Smart Arrays

A potentially very important class of value-semantic classes are *smart arrays*. We have given numerous examples of these already. One can use smart arrays to write extremely modular, reusable, and expressive code, such as

```
f = mesh->Del()*T + b;
```

to represent an equation such as

$$\hat{f} = \nabla \cdot \mathbf{T} + \hat{b} \quad (\text{EQ 1})$$

In this code, we use smart arrays to represent vector and tensor fields, and the object `mesh` hides the discretization of the spatial domain into a finite difference grid, finite element set, or other mesh topology. Unfortunately, smart arrays are very difficult to implement efficiently [5] [9].

One problem with smart arrays is the presence of *aliasing ambiguities*. These occur because the array class must be implemented using a pointer to the actual data. The compiler often is unable to determine whether multiple pointers in its scope point to the same object. Aliasing ambiguities force the compiler to generate code that makes poor use of registers and caches, particularly on high-performance computers. For example,

```

class dblarray {
private:
    size_t length;
    size_t *reference_count;
    double *array;

    dblarray(double *d, size_t l) :
        length(l),
        reference_count(new size_t(1)),
        array(d)
    {}

public:
    friend dblarray operator*(const dblarray&,
                              const dblarray&);

    /* ... */
};

dblarray operator*(const dblarray& a,
                  const dblarray& b){
    const size_t l = a.length;

```

```

    assert(l == b.length); // turned off for release
    double *d = new double[l];
    for (register i=0; i<l; i++) d[i] = a.array[i] * b.array[i];
    return dblarray(d, l);
}

```

In general, the compiler must assume that the array pointed to by `d` might overlap the array pointed to by `a.array` or `b.array`. As a result, it will generate code that ensures that `d[i]` is updated in memory before `a.array[i+1]` or `b.array[i+1]` are fetched. This prevents effective use of registers, pipelines, and caches and reduces performance on a wide variety of high-performance computers. On vector supercomputers, the performance may suffer by a factor of *~40-80!*

In many cases, this problem can be solved by the introduction of a suitable pointer qualifier to indicate that aliases are not allowed [6] or by permitting the compiler to assume that the pointer returned by the `new` operator is unaliased [4]. The introduction of a new pointer qualifier would be an extension to the language and has been rejected for the present by the ANSI C++ committee. However, the assumption that `new` returns an unaliased pointer appears to be justified by the current language of the ANSI working draft.

Another difficulty with smart arrays is the *proliferation of temporaries*. This arises because the current C++ grammar forces each operation in an array expression to be fully evaluated in sequence. In other words, an expression such as

```
y = a + b*x;
```

translates into code that fully evaluates `b*x`, storing the result in a temporary array `t`; then fully evaluates `a + t`, storing the result in a second temporary `tt`; then “evaluates” `y = tt`. This is a rather simple-minded way to evaluate the expression. It results in excessive memory usage and hinders efficient use of registers and caches. Furthermore, if temporaries are not destroyed at the end of expressions, large amounts of memory may be tied up in dead temporaries by the time the flow of control reaches the end of a program block. In effect, C++ mandates that this computation be decomposed first by operation and then by array element. On almost any platform, it is more efficient to decompose first by array element and then by operation.

Optimization across the above expression requires that the overloaded operator calls be inlined. Most of today’s translators cannot inline functions containing loops. However, the GNU C++ compiler claims this capability, suggesting that this is not an insurmountable obstacle [12]. With full inlining of operator functions, the expansion of the expression might be:

```

{
    double tmp1, tmp2;
    const size_t l = b.length;
    double *d = new double[l];
    for (register i=0; i<l; i++) d[i] = b.array[i] * x.array[i];
    tmp1.length = l;
    tmp1.reference_count = new int(1);
    tmp1.array = d;
    const size_t m = a.length;
    double *e = new double[m];
    for (register j=0; j<m; j++) e[j] = a.array[j] + tmp1.array[j];
    tmp2.length = m;
    tmp2.reference_count = new int(1);
    tmp2.array = e;
    if (!--*y.reference_count){
        delete y.reference_count;
        delete[] y.array;
    }
    y.length = tmp2.length;
    *(y.reference_count) = tmp2.reference_count++;
    y.array = tmp2.array;
    if (!--*tmp2.reference_count){
        delete tmp2.reference_count;
    }
}

```

```

    delete[] tmp2.array;
}
if (!--*tmp1.reference_count){
    delete tmp1.reference_count;
    delete[] tmp1.array;
}
}

```

Ideally, this should subsequently be transformed to

```

{
    const size_t m = a.length;
    if (!--*y.reference_count){
        delete y.reference_count;
        delete[] y.array;
    }
    y.length = a.length;
    y.reference_count = new int(1);
    y.array = new double[m];
    for (register i=0; i<m; i++)
        y.array[i] = a.array[i] + b.array[i] * x.array[i];
}

```

One might rely on improved inlining and very smart optimizers to generate efficient smart array code in this way. However, such compilers will likely be expensive to develop and support.

A second solution is to rely on sophisticated class definitions. Techniques such as return-by-reference or deferred expression evaluation can cut memory usage and yield acceptable computational efficiency for very large arrays [8], [10]. However, the overheads of some of these methods are prohibitive for smaller arrays.

A third solution is to extend the C++ grammar to permit overloading of entire parse trees. This looks attractive, but one encounters a combinatoric explosion of possible parse trees when more than a few operations are combined. Each requires its own overload function. However, even the ability to overload the simplest and most common operator combinations would be helpful. We are aware of at least one extended C++ translator which supports combination operators, and this approach will undoubtedly be explored further [1].

Finally, one can find a way to introduce array syntax into C++ at an atomic level. This requires built-in array types, which could look like value-semantic classes to avoid changing the core language [4]. Programmers could use such classes as building blocks for their own, specialized, smart array classes. This is the solution we advocate.

## **Built-In Array Classes**

Arrays are used in a wide variety of contexts. This implies that built-in array classes should be as low-level as the required optimization characteristics permit. The array classes that have been proposed to the ANSI C++ language standard committee are a representation of the mathematical concept of an ordered set of values.

These array classes are manipulated using a large set of overloaded operators and functions. All binary operations between arrays are element-by-element operations. Assignment is non-conforming, meaning that an array on the left-hand side of an assignment will be resized, if necessary, to conform to the right-hand side. A number of low-level topological operations are provided to facilitate the use of these classes as building blocks for higher-level classes.

The operations and functions chosen for the array classes make it possible to code any single-loop array operation as an expression. For example,

```

class Vector : private dblarray {
private:
    Vector(const dblarray& a) : dblarray(a) {}
}

```

```

public:
    Vector(void){}
    Vector(size_t n) : dblarray(n) {}

    friend Vector operator+(const Vector&, const Vector&);
    friend double operator*(const Vector&, const Vector&);
    /* ... etc ... */
};

Vector operator+(const Vector& a, const Vector& b){
    return Vector((dblarray&)a + (dblarray&)b);
}

double operator*(const Vector& a, const Vector& b){
    return (a*b).Sum();
}

```

However, many important array operations cannot be written as single-loop operations. The most important of these is matrix multiplication, which is normally written as a triple-loop operation that can be “unrolled” into a double-loop operation:

```

class Matrix : private dblarray {
private:
    size_t n;

public:
    Matrix(void){}
    Matrix(size_t nn) : dblarray(nn), n(nn) {}

    double Element(size_t r, size_t c){
        return dblarray::operator[](r + N*c);
    }

    friend Matrix operator*(const Matrix&, const Matrix&);
};

Matrix operator*(const Matrix& a, const Matrix& b){
    const size_t N = a.n;
    assert(N == b.n);
    Matrix rtn(N);
    for (register i=0; i<N*N; i++){
        size_t r = i%N, c = i/N;
        rtn[i] =
            (a.gather(r, N*(N-1)+r, N)*b.gather(c*N, (c+1)*N)).Sum();
    }
    return rtn;
}

```

In this example, the innermost loop has been replaced with a `dblarray` expression. The first call to `gather` selects the `r`th row of `a` and the second call to `gather` selects the `c`th row of `b`.

There is a bewildering variety of double-loop operations. Most of these operations appear in the context of matrix algebra, and there are literally dozens of distinct patterns of sparseness for which matrix operations might be written. This suggests that the line should be drawn at single-loop operations.

Although we have experimented with different implementations of the array classes described in [4], we are not in the business of compiler development and have not been able to experiment with built-in compiler support for these classes. We would like to see compiler developers explore this approach to solving the smart array problem.

## **Other Aspects of C++**

In this section, we discuss some miscellaneous aspects of C++ from the point of view of numerical programming.

We have only begun to explore the potential of templates to date. However, we expect that templates will simplify the task of developing families of specialized value-semantic classes. We are pleased to find that our C++ compiler ensures that identical template instantiation in separate translation units does not result in multiple copies of large template functions, as we initially feared would be the case.

We have had no experience with exception handling, since this feature is not yet widely available, but fear that the necessity of supporting stack unwinding will result in unacceptable run-time overheads. Implementations of exception handling should ensure that no such overheads are incurred if exceptions are not thrown.

A major challenge with our larger codes has been the implementation of persistent polymorphic objects. We expect that run-time type identification will provide features that support reasonable idioms for persistent object storage. However, we have had no experience to date, since RTTI is not yet available.

One of our major concerns is the lifetime of temporaries. If temporaries are not destroyed at the end of expressions, it is possible for large amounts of memory to be tied up in dead temporaries by the time the flow of control reaches the end of a block. We prefer immediate destruction, but recognize that this will break much existing code. We believe that destruction at the end of expressions is a reasonable compromise.

## **Conclusions**

Experience shows that both polymorphic classes and value-semantic classes are of value in numerical programs. C++ is unique among popular programming languages for its ability to support both. Our experience in programming large simulation codes in C++ suggests that codes should be structured as a set of polymorphic classes which exchange messages, represent internal states, and perform computations using value-semantic classes.

Efficiency concerns in such codes will be focused on the value-semantic classes. We believe that code using relatively simple value-semantic classes can be optimized satisfactorily using existing compiler technology. However, for the important case of smart arrays, the current compiler technology is inadequate. We advocate the adoption of generic array classes, which can be implemented as built-in types, to fill this gap.

Table 1: Polymorphic Classes vs. Value-Semantic Classes

	Polymorphic Class	Value-Semantic Class
Hierarchy mechanism	Inheritance	Conversion
Computational Interface	Message Passing	Algebra
Interface Mechanism	Member Functions	Overloaded Operators
Degree of Complexity	High	Low
Number of Specializations	Numerous	Few
Computational Efficiency	Not Critical	Critical

Figure 1. Illustration of Model Class Hierarchy

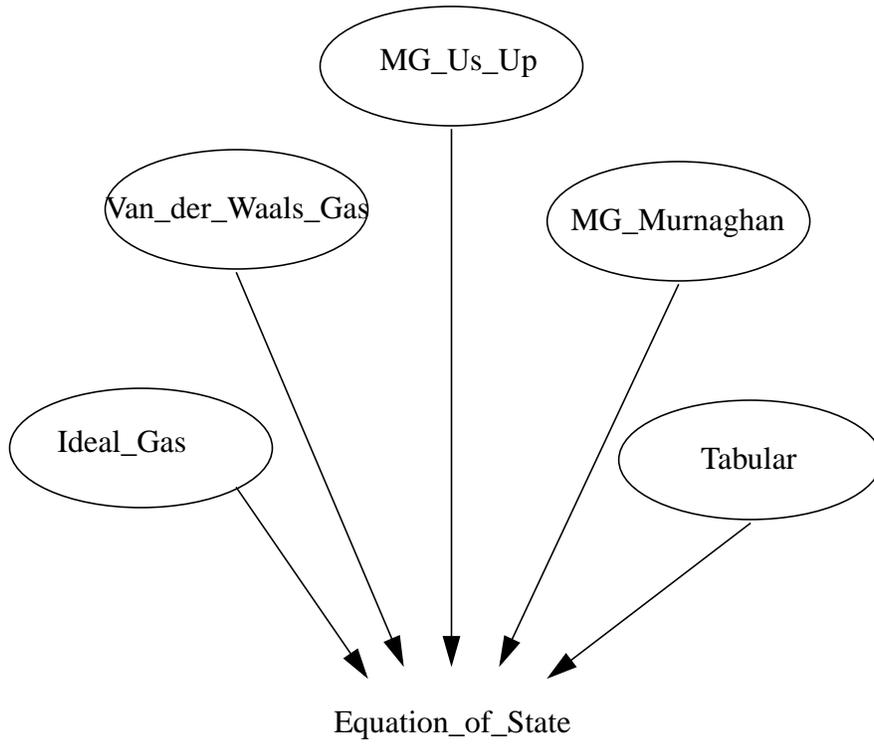
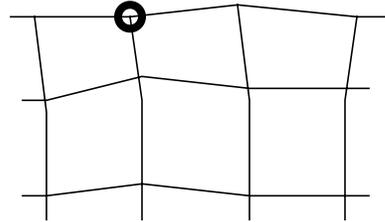
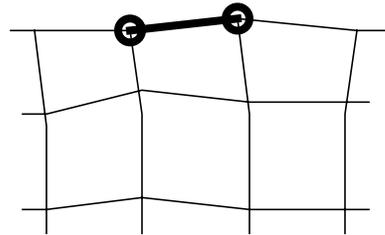


Figure 2. Illustration of Finite Element Boundary Classes

```
class Node_Set
{
  protected:
    size_t num_nodes;
    int *node;
    /* ... */
};
```



```
class Side_Set :
  public Node_Set
{
  protected:
    size_t num_faces;
    int *face_node[2];
    /* ... */
};
```



```
class Surface :
  public Side_Set
{
  protected:
    int *side_neighbor[2];
    /* ... */
};
```

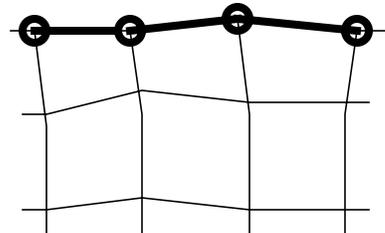


Figure 3. Boundary Class Hierarchy

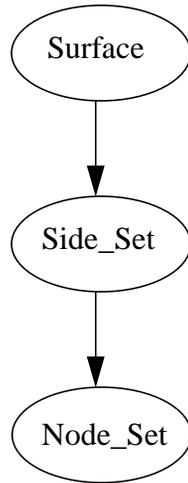


Figure 4. Mesh Class Hierarchy

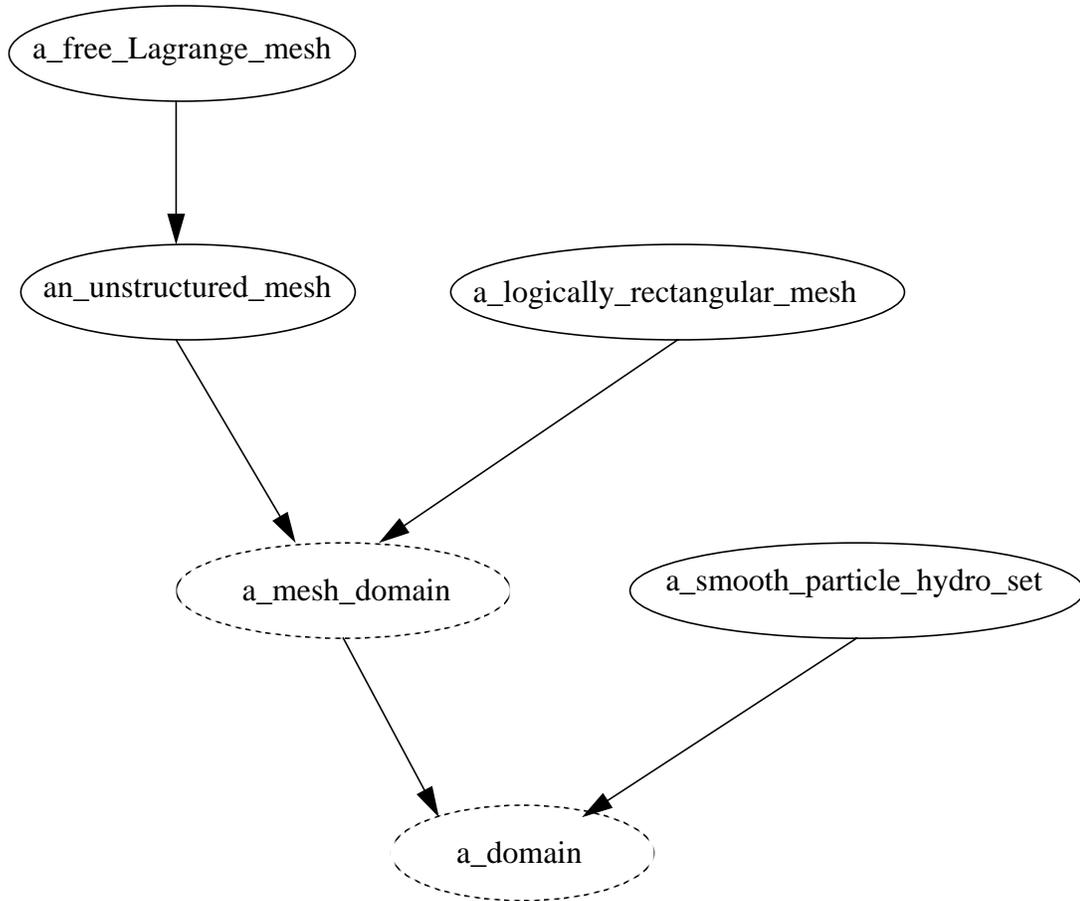
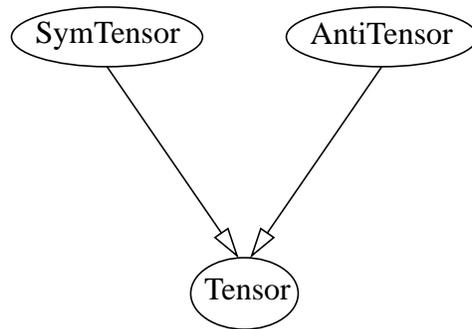


Figure 5. Illustration of the Conversion Hierarchy in the PHYSLIB Library



## References

- 1 *ARC++ User's Manual*. ARSoftware Corporation, Landover, Maryland, 1993.
- 2 Booch, G., *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc, Redwood City, California, 1991.
- 3 Budge, K.G., "PHYSLIB: A C++ Tensor Class Library." SAND91-1752, Sandia National Laboratories, Albuquerque, New Mexico 1991.
- 4 Budge, K.G., "Proposal for a Numerical Array Library." ANSI X3J16-93-0042/ISO WG21-N0249, March, 1993.
- 5 Budge, K.G., Peery, J.S., and Robinson, A.C. "High-Performance Scientific Computing Using C++." *Proceedings of the 1992 USENIX C++ Technical Conference*, August 1992.
- 6 Holly, M. "New Keyword for C++: Restrict." X3J16-92-0057 / WG21-N0134, June 1992
- 7 Murray, R. "C++ Tactics." AT&T Bell Laboratories, 1990.
- 8 Quinlan, D., "P++, an Architecture-Independent Software Development Environment." *Proceedings of the Copper Mountain Conference on Iterative Methods*, Copper Mountain, CO, April 9-14, 1992.
- 9 Robinson, A.C., et al., "Massively Parallel Computing, C++ and Hydrocode Algorithms." *ASCE 8th Conference on Computing*, June 1992.
- 10 Schutt, J.A., private communication, 1993.
- 11 Shopiro, J., private communication, 1991.
- 12 Stallman, R.M. *Using and Porting GNU CC*. Free Software Foundation, Inc., Cambridge, Massachusetts, 1989.
- 13 Stroustrup, B. and Ellis, M. *The Annotated C++ Reference Manual*, Addison Wesley, Reading, Massachusetts, 1990.
- 14 Verner, D. "Developing Generic Classes for Finite Element and Finite Difference Problems." Master's thesis, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, New Mexico, 1993.
- 15 Wong, M.K., and Fang, H.E., "MatResLib: A Reusable, Object-Oriented Material Response Library." *Proceedings of the First Annual Object-Oriented Numerics Conference*. Rogue Wave Software, Inc., Corvallis, Oregon, 1993.